AFRL-RI-RS-TR-2019-126

INFORMATION DRIVEN, ADAPTIVE DISTRIBUTED PLANNING

THE UNIVERSITY OF TULSA

JUNE 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-126 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

WILLIAM DAVID LEWIS
Work Unit Manager

/ S /

JULIE BRICHACEK
Chief, Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2019	FINAL TECHNICAL REPORT	SEP 2016 – DEC 2018

4. TITLE AND SUBTITLE

INFORMATION DRIVEN, ADAPTIVE DISTRIBUTED PLANNING

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-16-1-0253

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Roger Mailler and Rose Gamble

5d. PROJECT NUMBER
S2MY

5e. TASK NUMBER
RA

5f. WORK UNIT NUMBER
ST

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Tulsa
800 South Tucker Drive
Tulsa, OK 74104

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISC
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-126

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report details our approach to combining dynamic, distributed constraint reasoning with machine learning techniques and adaptive response strategies. By combining these technologies, we built a system that can 1) develop robust, adaptable mission plans 2) exploit knowledge learned through prior interactions with our adversary, and 3) autonomously and dynamically alter its behavior during mission execution to improve the likelihood of a successful outcome. This system has been thoroughly tested in the ATE2 and ATE3 simulators that were provided by AFRL/RI against four increasingly difficult milestones.

15. SUBJECT TERMS

Task allocation, intelligent path planning, ISR management

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM DAVID LEWIS**
U	U	U	UU	56	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-7707**

Table of Contents

List of Figures

List of Tables

1.0 Summary

Robert Burns once wrote: “The best-laid plans of mice and men oft go astray.” Nowhere is this statement truer than when planning missions in an uncertain, adversarial environment. Despite our best efforts and hundreds of years of experience, planning is still an arduous task whose results often dictate the outcome before the first action is ever taken. However, as Burns points out, even the best laid plans must be able to adapt to unforeseen circumstances to ensure that they succeed.

The overall goal of this project was to develop new technologies to dynamically control and coordinate multiple Unmanned Aerial Vehicles (UAVs) so they can accomplish their missions while the enemy is attempting to deny them access and prevent them from communicating. This report details our approach to addressing this complex problem by augmenting dynamic, distributed constraint reasoning with machine learning techniques and adaptive response strategies. By combining these technologies, we built a system that can 1) develop robust, adaptable mission plans, 2) exploit knowledge learned through prior interactions with an adversary, and 3) autonomously and dynamically alter its behavior during mission execution to improve the likelihood of a successful outcome. This system has been thoroughly tested in the ATE[2] and ATE[3] simulators that were provided by AFRL/RI against four increasingly difficult milestones.

Clearly, there were several key challenges that needed to be addressed in order to build this system. First, although we have already developed a system that uses constraint reasoning to solve ISR allocation problems and can leverage that experience, allocating an asset is very different from developing a specific plan. To rectify this distinction, we developed a mission planner that represents threats and goals as dynamic, geo-temporal constraints. We designed the mission planner using MTL robustness and A* to create robust plans that allow flexibility in the autonomous decision making of the agent. This flexibility is a particularly important feature because the UAVs cannot always predict the enemy’s actions and may not be able to communicate when it needs to alter its mission. Second, nearly all distributed protocols assume that communications are perfect. Our system explicitly reasons about communications, determines if it is necessary to communicate, and potentially adapts the mission in response. We developed a simple, self-adaptive technique with multiple layers as a basis for our coordination process. Finally, we developed learning techniques that are able to dynamically generate and update the geo-temporal constraints. We utilized a layer-learning approach that is formed using Kalman filters, Markov Decision Processes (MDPs), and Inverse Reinforcement Learning (IRL). In contrast to traditional reinforcement learning, our approach learns the enemy’s policies and optimizes our actions using distributed constraint optimization methods. Thus, the system is not reacting to the enemy, but rather taking actions that exploit weaknesses in their reactions. By combining these learning methods, the resulting system is capable of online adaptation as well as offline learning.

The result of this effort includes more than just a potential solution to a single problem. The techniques and protocols developed during this project enable smooth system planning and adaptation, increase the mission robustness of the autonomous vehicles in adversarial environments, and minimize vehicle loss during mission execution by learning the enemy’s behavior. This project provides a deeper understanding of how to integrate multiple AI technologies into a resilient system that acts autonomously in a real-world setting.

2.0 Introduction

Performing Intelligence, Surveillance, and Reconnaissance (ISR) in uncertain, adversarial environments is a complex task that involves sophisticated pre-planning combined with dynamic adaption during mission execution. This is particularly true when mission success is dependent upon coordinating multiple, heterogeneous autonomous assets against an enemy that is intelligent, capable, and highly mobile. Succeeding in these environments does not necessarily require technological or numeric superiority if strategic and/or tactical patterns can be identified and exploited. Therefore, it is imperative that systems of the future be designed to recognize and adapt to these patterns to increase their resiliency and maximize the probability of a successful outcome.

The ISR mission planning domain has several key features that must be taken into consideration. First, assets must be able to act intelligently and autonomously to minimize the need for human action. Second, communication is limited to prevent detection and hostile electronics warfare. Third, the system must be able to handle 100s of assets doing 1000s of missions. Fourth, the enemy is acting intelligently to prevent missions from completing causing the environment to constantly change.

This project combines dynamic, constraint-based coordination with learning technology and adaptation strategies to achieve the overall objective of ISR missions in adversarial environments. The goals of this fusion were to (1) develop distributed technologies that can identify patterns in behavior (**learn**) and (2) exploit these patterns to devise coordinated strategies to maximize goal completion with minimal cost (**adapt**). These goals naturally led to several important research questions that we addressed in this project:

1. **How do we design a system that intelligently acquires and delivers relevant information to the right asset to support the learning and coordination tasks in a dynamic, communications degraded environment?** Allocating multiple UAVs to perform sub-tasks cooperatively toward achieving a global mission objective is a challenge covered under different research areas. The problem becomes harder when the knowledge about the tasks and the risk distribution is imperfect with communication impairments. In addition to that, the capability of each UAV is restricted according to its sensor type, fuel capacity, mission deadline, and weapon ammunition. The goal is to design a coordination strategy allowing the UAVs to reason about each other and make decisions when the communication is degraded.
2. **What learning technologies provide the most appropriate and efficient strategies given the bounded rationality of the agents and the changing behavior of the enemy?** To enhance the robustness of the generated plans for our UAVs, we need to identify patterns in the movement of individual threats and targets as well as learning patterns of enemy reactions to the actions taken by the friendly forces. Our objective is to identify the specific technologies that would perform the fastest and have the most accurate responses given the data. After anticipating the behaviors of our adversaries, the goal becomes about computing plans that would avoid the anticipated models to increase the safety of UAVs.
3. **How can learned knowledge be used to identify opportunities for generating coordinated UAV activities that accomplish tasks, which would have been impossible**

or prohibitive to accomplish otherwise? Cooperative, distributed planning is a complex problem where each UAV develops an individual, local plan that is refined while coordinating with other UAVs to avoid potential conflicts and use cooperative opportunities for improved completion of mission objectives. The coordination objective is to maximize the availability, timeliness, and accuracy of information in a rapidly-changing, uncertain, and possibly hostile environment. To achieve high-level goals, such as multi-modal data gathering, under these conditions, UAVs must not only be able to reason introspectively (i.e. what actions and information are needed to achieve the local plan), but also extrospectively (i.e. what actions and information are needed for others to achieve their local plans).

3.0 Methods, Assumptions, and Procedures

3.1 Characterizing an environment

Our overall approach is to formulate this problem as a dynamic, distributed constraint optimization problem (DynDCOP). This formulation is ideal for the ISR mission planning domain because the assets can act autonomously to adapt their plans, communications can be limited to only the most important information, 1000s of assets can be simultaneously controlled without a central point of failure, and it can to **adapt** to both fast and slow changes in the assets, goals, and constraints [1].

The distributed constraint satisfaction problem (DCSP) [2] and distributed constraint optimization problem (DCOP) [3] have been used as common formalisms to describe problems including distributed resource allocation [4], distributed heuristic search [5], and distributed interpretation [6]. Formally speaking, a Distributed Constraint Satisfaction/Optimization Problem (DCSP/DCOP), $\boldsymbol{P} = \langle V, A, D, f \rangle$, consists of the following [2]:

- A set of *n* variables, $V = \{x_1, \dots, x_n\}$.
- A set of g agents, $A = \{a_1, \dots, a_g\}$.
- A set of discrete, finite domains for each variable $D = \{D_1, \dots, D_n\}$.
- A set of utility functions $f = \{f_1, \dots, f_m\}$ where each $f_i(d_{i,1}, \dots, d_{i,j})$ is a function $f_i: D_{i,1} \times \cdots \times D_{i,j} \mapsto \mathbb{R} \cup \infty$ where the functions take in a subset values of the variables and return a utility.

The task is to find an assignment $S^* = \{d_1, \dots, d_n | d_i \in D_i\}$ such that the global utility, called *F*, is maximized. The function *F* can be any associative, commutative, monotonic aggregation function defined over a totally ordered set of values, with min and max elements. *F* is often defined as $F(S) = \sum_{i=1}^{m} f_i(S)$.

Building on our prior work, each goal (surveilling a target) is modeled as a variable and potential mission assignments (asset/plan combinations) as its domain. We also define a set of unary constraints that represent the assets' capabilities and a set of binary constraints that prevent an asset from being assigned two different missions during the same time.

For dynamic settings, we consider how the problem is changing by introducing a function $\Delta: P_t \mapsto P_{t+1}$ that maps a problem at time *t* to a new problem at *t*+1 [7]. This function creates a

sequence of problems: $\boldsymbol{P} =< P_0, P_1, \ldots, P_n >$ where P_i is a static problem that entails a set $S_i = \{s_i^1, \ldots, s_i^m\}$ of assignments each with its own utility $U(s_i^k)$. The general form of this function is $P_i = P_{i-1} + f_i^a - f_i^r$ where f_i^a and f_i^r are the set of added and removed interrelationships [8]. This leads to a characterization of the rate of change of the problem as

$$\frac{dP}{dt} = \lim_{\Delta t \to 0} \frac{P_{t+\Delta t} - P_t}{\Delta t} = \frac{1}{\Delta t} \sum_{i=t}^{t+\Delta t} (|f_i^a| + |f_i^r|)$$

For this system to work, it now needs to consider the enemy. This is done by introducing a new set of dynamic constraints. The new dynamic, geo-temporal constraints (GTCs) represent areas that must be avoided in order for the mission to succeed. These constraints are dynamic, because we may not know the locations of all of the enemy assets and the assets change position over time. Formally, a dynamic GTC is a function $gtc(lat, lon, t) \rightarrow \mathbb{R} \cup \infty$. Here the function takes a latitude and longitude location at a particular time *t* and returns a cost associated with being in that place at that time. We use the set of GTCs to compute the utility of a particular mission by iterating over the mission plan and computing the cost of the individual GTCs. To precisely and simply specify our GTC, we use Metric Temporal Logic (MTL), [9] which is typically used in offline verification and online control of hybrid systems.

To address this problem, we have set some assumptions regarding the environment, targets, and enemies. The targets have a range called the *terminal range*. For each problem instance, the goal is for the UAV (from now on we call it drone) to eventually be located within terminal range of its assigned target. The environment is non-deterministic, such that the drone has partial information about enemies prior the mission, including their geometry and capabilities, but the distribution of enemies is unknown until the drone traverses the world and uses its sensors for detection. The drone can perceive the environment around it within a circular region centered at the drone location where the radius of the circle is its vision range. The environment under investigation has three types of static and mobile enemies with different risk ranges:

1. **Radars**: able to see and detect the drone within its *visual range* (VR).
2. **Jammers**: able to jam the drone's communications if the drone gets in its *jamming range* (JR).
3. **Killers**: able to shoot missiles when the drone gets inside its *weapon range* (WR).

3.2 General Approach: Information Driven, Adaptive Distributed Planning

We developed a framework with five major components: Mission Planner Module (MPM), Mission Execution Module (MEM), Coordination Module (CM), and Pattern Learning Module (PLM). These modules represent the combination of knowledge representation and planning, system control, machine learning, and dynamic, distributed constraint reasoning. Figure 1 shows our framework.

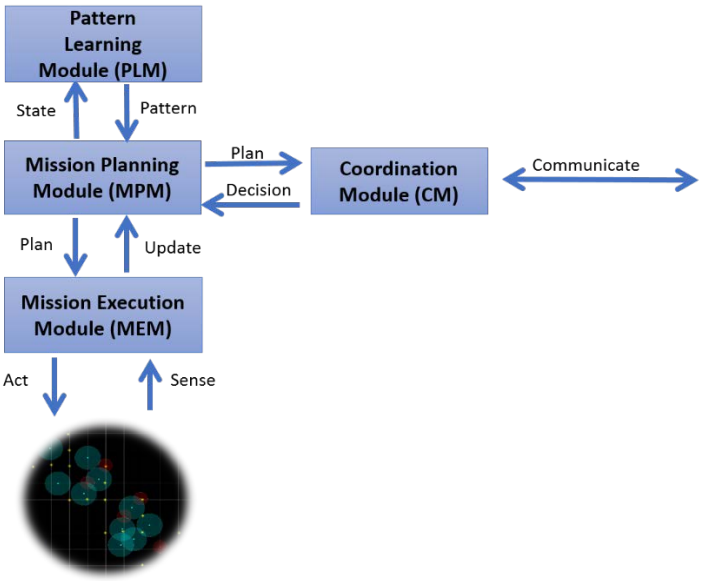

Figure 1: The developed framework

3.2.1 Mission Planning Module: MTL Robustness for Path Planning with A*

We built our MPM by developing a robustness function using the robustness theory of Metric Temporal Logic (MTL) [9] to maximize the drone safety while satisfying mission constraints. MTL robustness can be defined as the upper-bounded perturbation that the drone can tolerate without changing its Boolean truth value with respect to its mission specification expressed in MTL [9]. Specifically, if an MTL specification φ evaluates to a positive robustness ε, then the specification is true, i.e., satisfied and, moreover, the path points can tolerate perturbations up to ε and still satisfy the specification. Similarly, if ε is negative, then the path point does not satisfy φ and all the other points that remain within the open tube of radius $|\varepsilon|$ also do not satisfy φ.

Our approach to address the reach-while-avoid-when-possible problem has two main steps. First, the mission constraints are simply and concisely expressed using MTL specifications [10]. Secondly, we developed Robust A* (RA*) by extending A* with two modifications: (i) a soft modification of the objective function to include the MTL robustness function, and (ii) a hard modification of the algorithm logic to exclude the non-robust positions from the search space. The robustness function is used to guide the node expansion in RA* and dynamically create a safety margin around adversarial assets using the drone resources. In addition, RA* creates a robust neighborhood around the generated path using the robustness degrees of the path points [9, 11]. The robust neighborhood provides a set of valid trajectories that can be robust enough for the drone to autonomously react to moving enemies or fuel loss without conducting a re-planning process.

Figure 2 displays a closed-loop process that ends when the drone reaches the target. The RA* generates the robust neighborhood while Robust_Heuristic_Search (RHS) adjusts the path when new information is detected without the need for re-planning.

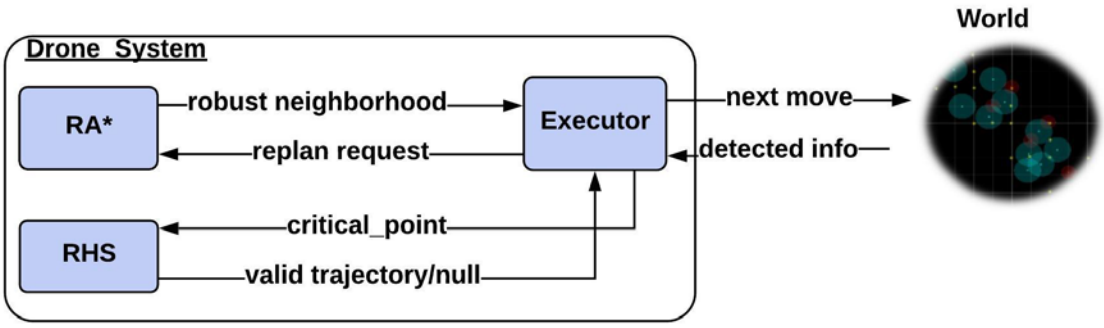

Figure 2: The system model for MPM and MEM

The RA* algorithm creates a safety margin around the closest enemy to set the minimum acceptable risk at each path step. The safety margin size is computed using the robustness function to maximize the satisfaction of the mission constraints along the planned path. RA* keeps track of a robust neighborhood around the optimal trajectory to reduce re-planning attempts and increase drone resilience against moving enemies. The robustness measure returns positive values if the trajectory satisfies the specification and negative values otherwise. Intuitively, the robustness degree of a feasible path is the largest distance the drone can independently perturb and still maintain the feasibility of its current path. This defines a neighborhood around the original path such that any trajectory within this neighborhood is guaranteed to satisfy the specification but with a lower degree of robustness. When the drone detects a violation of its constraints, the RHS utilizes the current robust neighborhood to find a valid replaceable trajectory. When the whole neighborhood becomes invalid, re-planning needs to be executed by RA*. The objective is to find a neighborhood with a set of valid trajectories and an optimal path at the neighborhood's center instead of planning for a single path. However, when the algorithm can only find one feasible path then the robust neighborhood is collapsed into a single trajectory. A collapsed neighborhood occurs when the available resources approach their limits and, consequently, the drone must take some allowable risk to reach its target.

Definition 1 (MTL Syntax). Let AP be the set of atomic propositions and *I* be a time interval of R. The MTL φ formula is recursively defined using the following grammar [10]:

$$\varphi \coloneqq T|p|\neg\varphi|\ \varphi_1 \vee \varphi_2|\varphi_1 \wedge \varphi_2|\ \varphi_1 \mathcal{U}_I \varphi_2 \tag{1}$$

T is the Boolean True, p ∈ *AP*, ¬ is the Boolean negation, ∨ and ∧ are the logical OR and AND operators, respectively. $\mathcal{U}_I$ is the *timed until* operator and the interval *I* imposes timing constraints on the operator. Informally, $\varphi_1 \mathcal{U}_I \varphi_2$ means that φ_1 must hold until φ_2 holds, which must happen within the interval *I*. The implication (⟹), Always (□), and Eventually (◊) operators can be derived using the above operators.

Using the MTL syntax (**Definition 1**), we define the MTL specification of our problem of reach-while-avoid-when-possible as follows.

$$\varphi = \Diamond_{[0,timeConstraint]}\, q \wedge \Box_{[0,timeConstraint]}\, (\neg unsafe\,) \wedge \Box_{[0,timeConstraint]} \big((\, constrain_1 > threshold_1 \wedge \ldots \wedge \; constrain_n > threshold_n\,) \Longrightarrow \neg semiSafe\big) \quad (2)$$

This formula requires the drone to reach the target terminal q (i.e., liveness property) while always avoiding being inside *unsafe* areas (i.e., safety property). When the available $constraint_1$, $constraint_n$ are above their predefined thresholds, it must always stay away from the semi-safe areas (i.e., conditional safety property). Otherwise, the semi-safe areas need to be gradually receded to free up some space for the drone to maneuver in to reach its target. We define a path trajectory that satisfies the specification given in (2) to be a feasible trajectory. Otherwise, it is infeasible. For our problem domain, this specification would be:

$$\varphi = \Diamond_{[0,deadline]}\, q \wedge \Box_{[0,deadline]}\, (\neg WR \wedge \Box_{[0,deadline]} \big((\, f > f_{min} \wedge t < t_{max}\,) \Longrightarrow \neg VR \wedge \neg JR\big) \quad (3)$$

To precisely capture the MTL formula, each predicate $p \in AP$ is mapped to a subset of the metric space S. Let $\mathcal{O}: AP \longrightarrow \mathcal{P}(S)$ be an observation map for the atomic propositions. The Boolean truth value of a formula φ with respect to the trajectory *s* at time *t* is defined recursively using the MTL semantics directly reproduced as stated in [10]:

$$(s,t) \coloneqq T \iff T$$

$$\forall\, p \in AP, (s,t) \coloneqq \mathcal{O}\, p \iff s_t \in \mathcal{O}(p)$$

$$(s,t) \coloneqq \mathcal{O}\, \neg\varphi \iff \neg(s,t) \coloneqq \mathcal{O}\, \varphi$$

$$(s,t) \coloneqq \mathcal{O}\, \varphi_1 \vee \varphi_2 \iff (s,t) \coloneqq \mathcal{O}\, \varphi_1 \vee (s,t) \coloneqq \mathcal{O}\, \varphi_2$$

$$(s,t) \coloneqq \mathcal{O}\, \varphi_1 \wedge \varphi_2 \iff (s,t) \coloneqq \mathcal{O}\, \varphi_1 \wedge (s,t) \coloneqq \mathcal{O}\, \varphi_2$$

$$(s,t) \coloneqq \mathcal{O}\, \varphi_1 \mathcal{U}_I \varphi_2 \iff \exists t' \in t + I.\, (s,t') \coloneqq \mathcal{O}\, \varphi_2 \wedge \forall\, t'' \in (t,t'), (s,t'') \coloneqq \mathcal{O}\, \varphi_1$$

In our problem domain, we have 6 atomic propositions including time, fuel, risk ranges (VR, JR, and WR), and the target. To properly use the observation map semantics in the problem domain, we compute time and fuel in terms of distance metric d, which is the Euclidian distance between points p_1, p_2, as:

$$d(p_1,p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (4)$$

By using the drone's velocity v, and the fuel consumption rate crt (fuel per distance), we calculate the time and fuel in terms of distance as follows:

$$t = \frac{d}{v} \quad (5)$$

$$f = d \times crt \quad (6)$$

To formally measure the robustness degree of φ (2) at the trajectory point *s* at time *t*, the robustness semantics of φ is recursively defined as taken directly from [9]:

$$
\begin{aligned}
[\![T]\!](s,t) &:= +\infty \\
[\![p]\!](s,t) &:= Dist_d(s(t), \mathcal{O}(p)) \\
[\![\neg\varphi]\!](s,t) &:= \neg[\![\varphi]\!](s,t) \\
[\![\varphi_1 \vee \varphi_2]\!](s,t) &:= [\![\varphi_1]\!](s,t) \sqcup [\![\varphi_2]\!](s,t) \\
[\![\varphi_1 \wedge \varphi_2]\!](s,t) &:= [\![\varphi_1]\!](s,t) \sqcap [\![\varphi_2]\!](s,t) \\
[\![\varphi_1 \mathcal{U}_{[l,u]} \varphi_2]\!](s,t) &:= \bigsqcup_{j=t+1}^{t+u} \left([\![\varphi_2]\!](s,j) \sqcap \prod_{k=t}^{t-1} [\![\varphi_1]\!](s,k) \right)
\end{aligned}
$$

where $\sqcup$ stands for maximum, $\sqcap$ stands for minimum, $p \in AP$, and $l, u \in \mathbb{N}$. The robustness is a real-valued function of the trajectory point s with the following important property stated in Theorem 1.

Theorem 1 [9]: For any $s \in S$ and MTL formula φ, if $[\![\varphi]\!](s,i)$ is negative, then s does not satisfy the specification φ at time i. If it is positive, then s satisfies φ at i. If the result is zero, then the satisfaction is undefined.

By maximizing the robustness degree $[\![\varphi]\!]$, we can compute the control inputs (direction, velocity) over the finite set of input sequences that provide us with a path solution to a given problem instance, assuming that there is at least one feasible path. The generated sequence of inputs can be simply considered as the sequence of path points, i.e., trajectory (s,t) to the target that satisfy φ by having positive robustness degree $[\![(s,t)]\!] > 0$. The larger $[\![(s,t)]\!]$, the more robust the trajectory is to a perturbation of φ. In other words, trajectory s can be disturbed at time t while $[\![\varphi]\!]$ decreases, but remains positive. Consequently, the robustness degree $[\![\varphi]\!]$ of each path trajectory creates a robust neighborhood around the trajectory that provides a set of trajectories with less $[\![\varphi]\!]$, but still satisfy the original φ.

The Signed Distance, $Dist_d$, is a domain-specific function that must be defined to reflect the domain properties [9]. In this project, we define three functions to measure the distance from the propositions of the target, the enemy set, and the resource limits (Figure 3). The target symbol represents the target terminal while the blue circle is the drone. The red circle surrounded by multiple circles represents the enemy, where cyan, orange, and red circles are the VR, JR, and WR, respectively.

Definition 2 (Target $dist$ function): Given that q is the target terminal of drone p and r is its range, the $dist$ between q and p at time i is defined as

$$dist(p_i, q) = d(p_i, q) - r \tag{7}$$

Definition 3 (Enemy $dist$ function): Let p be the drone, X be the enemy set, and $\theta \geq 0$ be the enemy speed. Then $dist$ between p and X at time i is defined for each risk range as follows:

$$dist_{VR}(p_i, X) = \min_{0 \leq j \leq |X|} \mathrm{d}(p_i, x_j) - (x_j.VR + \theta) \tag{8}$$

$$dist_{JR}(p_i, X) = \min_{0 \leq j \leq |X|} \mathrm{d}(p_i, x_j) - (x_j.JR + \theta) \tag{9}$$

$$dist_{WR}(p_i, X) = \min_{0 \leq j \leq |X|} \mathrm{d}(p_i, x_j) - (x_j.WR + \theta) \tag{10}$$

With respect to the target q, $dist$ is defined as the distance from the drone to the closest edge of the region defined by the target's terminal range. On the other hand, the enemy $dist$ function is evaluated with respect to the drone p and the set of enemies X to a triple that represents the distances to the range of the closest enemy x to p (Figure 3). In addition, we use the velocity of a given enemy θ such that each risk range is extended with the enemy's velocity as represented by dotted circles around colored enemy ranges (Figure 3). The terminal range for a moving target must be shrunk, rather than extended, by the velocity of the target, assuming that the target is running away from the drone.

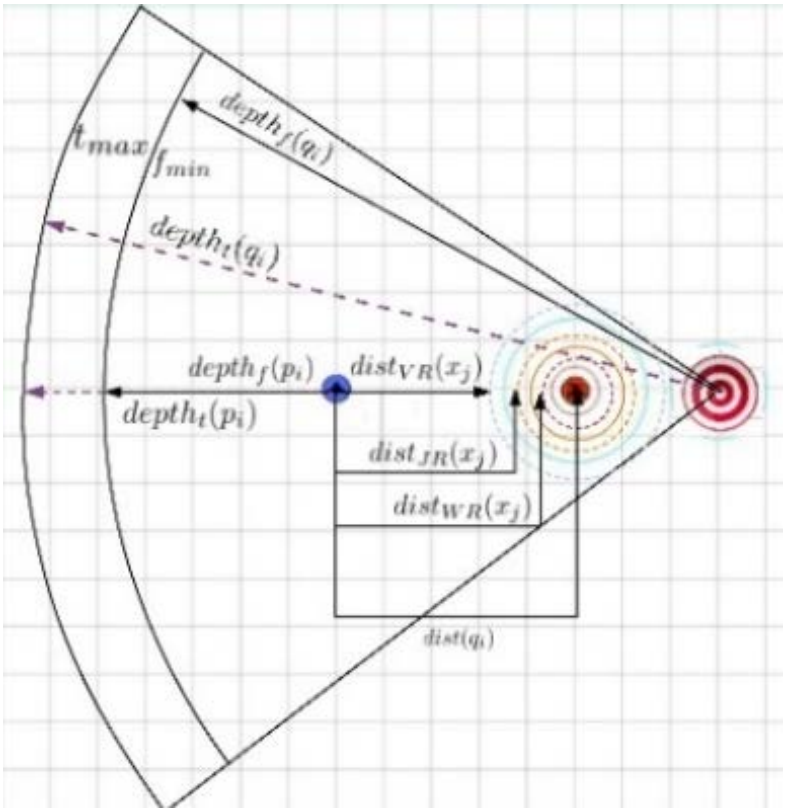

Figure 3: The structure of the problem domain

Lastly, we define a depth function to measure the distance between the current position of the drone and its resource limit. Each drone has a pre-specified amount of fuel and time to reach its target. We assume that each mission starts at time 0 with fuel f_{max} and each drone has time t_{max}, the deadline, to reach its target. To calculate the robustness of a given configuration of drones with allotted fuel capacities, fuel consumption rates, and deadlines, we redefine each constraint in terms of distance using equations (5) and (6). Given that a drone moves with velocity v, we define two regions centered at the target q with radius $v \times f_{min}$ and $v \times t_{max}$ to define the farthest positions that the drone could travel while still being able to reach its target. With these regions defined, we can define the function depth.

Definition 4 (Resource $depth$ function): Given that f_{min}, t_{max} are the resource thresholds, $depth_f$ and $depth_t$ functions for the drone p at time i are defined as:

$$depth_f(p_i) = \begin{cases} (f_i - f_{min}) - (d(p_i, q) - r) \times \text{crt} & if\ p_i \notin \mathcal{O}(f_{min}) \\ 0 & otherwise \end{cases} \quad (11)$$

$$depth_{\,t}(p_i) = \begin{cases} (t_{max} - t_i) - \frac{(d(p_i,q)-r)}{v} & if\ p_i \notin \mathcal{O}(t_{max}) \\ 0 & otherwise \end{cases} \quad (12)$$

The *depth*() function measures the distance to the closest edge of the region defined by a constraint centered on the target. It should be noted that the regions are 3D with respect to time. Therefore, the pizza slice shown in Figure 3 would be shrunk from the outer edge over time.

Using the *dist* and *depth* functions, the MTL robustness degree of φ in equation (3) can be point-wise computed at each position in the world state to solve the following path planning problem:

$$\rho\,(s_0, q) = min\ \sum c(s_i, s_{i+1}) - [\![(s_i, s_{i+1})]\!] \quad (13\text{-}1)$$

$$s.t. \quad 0 \le i < t_{max} \quad (13\text{-}2)$$

$$f_{min} \le f \le f_{max} \quad (13\text{-}3)$$

$$\epsilon_{min} \le [\![\rho\,]\!] \le \epsilon_{max} \quad (13\text{-}4)$$

$$\begin{aligned} \square_{[0,deadline]}\big(\neg WR_X(pos)\big) &= \neg\big(T\,\mathcal{U}_{[0,deadline]}\,\neg WR_X(pos)\big) \\ &\xrightarrow{[1,1]} = \neg\Big(\bigsqcup_{j=i+1}^{i+1}\big([\![T]\!](pos,i) \sqcap \bigsqcap_{k=1}^{i} \neg[\![WR_X]\!](pos,k)\big)\Big) \\ &= min\big(\infty, dist_{WR}(pos,X)\big) = dist_{WR}(pos,X) \end{aligned} \quad (14)$$

$$\begin{aligned} \square_{[0,deadline]}\Big(\big(f(pos) > f_{min} \wedge t(pos) < t_{max}\big) \implies \big(\neg VR_X(pos) \wedge \neg JR_X(pos)\big)\Big) \\ = \neg\Big(T\,\mathcal{U}_{[0,deadline]}\big(\big(f(pos) > f_{min} \wedge t(pos) < t_{max}\big) \wedge \big(\neg VR_X(pos) \vee \neg JR_X(pos)\big)\big)\Big) \\ \xrightarrow{[1,1]} = \Big(\bigsqcap_{j=i+1}^{i+1}\big([\![F]\!](pos,i) \sqcup \bigsqcup_{k=1}^{i}\big(\neg[\![f]\!](pos,k) \sqcup \neg[\![t]\!](pos,k)\big) \sqcup \big([\![VR_X]\!](pos,k) \sqcap [\![JR_x]\!](pos,k)\big)\big)\Big) \\ = max\Big(\neg depth_f(pos), \neg depth_l(pos), min\big(dist_{VR}(pos,X), dist_{JR}(pos,X)\big)\Big) \end{aligned} \quad (15)$$

where $c(s_0, s_i)$ is a cost function similar to the f function in A*. Equations (13-2) and (13-3) represent the resource limitation. ϵ_{min} and ϵ_{max} are the desired minimum and maximum robustness that make equation (13-4) an optional constraint in the problem. When $\epsilon_{min} > 0$, it enforces a minimum safety margin around enemies, while $\epsilon_{max} > 0$ attempts to limit the path length when the available resources are extremely large.

3.2.1.1 RA* Algorithm

The A* algorithm represents the world state as a grid map divided into squares. Each square is evaluated either as passable (safe and reachable), impassable when it is either occupied by one or more enemies, or unreachable when the drone does not have enough resources to reach it. However, the impassable squares that are occupied by the VR or JR of enemies can become passable when the drone cannot afford to avoid them given its limited resources.

RA* uses the same logic as the standard A*. The modified functions are presented in Algorithm 1. In line 3 of the *neighbors* function, the node is included into the search space only when the robustness is greater than or equal to zero. Our algorithm uses the MTL robustness to classify nodes in the gird into passable or impassible based on their satisfaction of the mission safety constraints. Only passable nodes are used to feed the open queue in RA*. The robustness of the liveness property is used in the *h* function to encourage the algorithm to expand its search towards positions closer to the target.

Assumption 1: If drone velocity is v and the max velocity of the enemy set is $v' = max_{x \in X} v_x$, then the drone is assumed to be faster $v > v'$ by at least 20%.

Assumption 2: If the drone vision range is VR and the enemy set's max vision range is $VR' = max_{x \in X} VR_x$, then the drone has bigger vision range $VR > VR'$.

Assumption 3: Given target q and enemy set X, q is not part of the enemy set $q \notin X$.

The robustness of the safety property ($\square_{[0,deadline]}$ ($\neg$WR)) is measured at each position of the search space since it must hold at all path points. To measure the robustness of the safety property for *pos*, we use the MTL robustness semantics with duration of [1, 1] and enemy set X is found in equation (14). In order to apply the robustness semantic, the *always* operator $\square$ is converted into the *Until* operator using the conversion rules in [11]. Then, the robustness becomes a minimum function of the robustness of *True* value and the *dist* function in equation (10). Since the robustness of *True* by the semantics is positive infinity, the robustness function becomes about the *dist* function of WR.

Line 4 in *robustness_safety* in Algorithm 1 defines the *dist* function using equation (10) to measure the distance from the WR of the enemy set. The positions inside the WR would have negative robustness, preventing the drone from taking paths through them. Positions with positive numbers are considered passable. Their robustness degree depends on how far they are from the boundary of the WR.

The conditional safety property $\left(\square_{[0,deadline]} \left((f > f_{min} \wedge t < t_{max}) \Longrightarrow \neg VR \wedge \neg JR \right) \right)$ evaluates the ability of the drone to avoid being seen or jammed by enemies considering its current time and fuel. It avoids an enemy JR and VR only when it has sufficient resources to do so, otherwise these areas are included in the search space as passable positions for the drone. The $depth$ functions, in equations (11) and (12), measure how far away the drone is from being out of time or fuel if it chooses to pass through position *pos*. The safety margin, i.e., the minimum robustness ϵ_{min}, is dynamically computed using the results of $depth_f$ and $depth_t$ (line 1 in *robustness_safety)*. The safety margin is decreased gradually with the time and fuel and becomes negative when either one of the resources starts approaching its limits. In lines 2 and 3 of function *robustness_safety*, the updated safety margin would be subtracted from the VR and JR to allow the drone pass through semi-safe areas. The *dist* functions in lines 2 and 3 return positive if the drone obeys the constraint of the minimum robustness and returns negative otherwise. The robustness of the conditional safety is computed in equation (15) using the MTL robustness semantics for the time duration of [1, 1] and enemy set *X*. The robustness function becomes about

finding the maximum values of the negative of $depth_f$ and $depth_t$ and the minimum of dist functions of equations (8) and (9).

The MTL robustness semantics in equation (15) is mapped into line 5 of *robustness_safety* function to decide if position *pos* is passable or impassable. Equation (15) would return negative values if the point *pos* is inside the jamming and vision ranges (JR, VR) of the closest enemy and the drone's time and fuel are above thresholds. On the other hand, it would return positive if and only if *pos* is outside the VR and JR of all enemies. In case the fuel or time are approaching their limits, equation (15) would always return zero when *pos* is inside VR or JR of enemies. According to Theorem 1, zero is undefined robustness, which RA* would accept only when there are insufficient resources to reach the target while avoiding areas with conditional safety property. By using this technique, MTL robustness allows runtime risk assessment of the VR and JR, which completely depends on the availability of resources.

Line 6 in *robustness_safety* computes the final robustness degree of *pos* as the min value of the safety and conditional safety properties. A negative value means the robustness is negative and *pos* is not inserted into the search space (line 3 of the *neighbors* function). Otherwise, the robustness degree is either positive or zero and *pos* is passable and considered for path planning search. By preferring paths with larger safety robustness in line 5 of the *neighbors* function, the algorithm generates robust paths to risky areas.

In *RA**, $\Diamond_{[0,deadline]}\, q$ evaluates the reachability of the target q from position *pos* in the *robustness_liveness* function. It depends on the *dist* function in equation (7), which returns a positive real number if *pos* is outside the terminal range of *q* and returns negative otherwise. Then, *h* in line 6 of *neighbors* function would have negative value when *pos* is inside the target area which decreases *f* in line 7. To guarantee that the algorithm expands positions with shorter distances to the target, the OPEN queue of A* is ordered based on the lowest (most robust) *f* values of searchable positions.

Algorithm 1 Functions needed for *RA Search**

function neighbors(p, ϵ_{min}, $\epsilon_{\max}$)
1- neighbors = neighbors_of(p,1)
2- **for** n ∈ neighbors
3- **if**(robustness_safety(n, X, ϵ_{min}, $\epsilon_{\max}$)>=0)
4- n.r=robustness_safety(n, X, ϵ_{min}, $\epsilon_{\max}$)
5- n.g = d(n,p)- n.r
6- n.h= robustness_liveness(neighbor, q)
7- n.f = n.g + n.h
8- n.parent = p
9- **return** neighbors
function robustness_safety(pos, X, ϵ_{min}, $\epsilon_{\max}$)
1- $\epsilon_{min} = \min(\epsilon_{min}, depth_f(pos) - \epsilon_{min}, depth_t(pos) - \epsilon_{min})$
2- $dist_{VR}(pos, X) = dist_{VR}(pos, X) - \epsilon_{min}$
3- $dist_{JR}(pos, X) = dist_{JR}(pos, X) - \epsilon_{min}$
4- safety =$dist_{WR}(pos, X)$
5-con_safety=

$$\max\begin{pmatrix} depth_f(pos), depth_t(pos), \\ \min\left(dist_{VR}(pos, X), dist_{JR}(pos, X)\right) \end{pmatrix}$$

6-**return** min (safety,con_safety, ϵ_{max})

function robustness_liveness(pos, q)

1- **return** $dist(pos, q)$

3.2.1.2 RHS

The loop function (Algorithm 2) of the drone (agent) moves the drone on the path and monitors robustness simultaneously. The drone's sensor detects enemies when they are within its visual range. The *violate_robustness* function (line 9 of loop) checks if the robustness of the current path is violated by newly detected information. It iterates over the current path points, re-evaluates the robustness of the safety property given available enemy information and the world state, and returns the first invalid path point, i.e., critical point *cp*. A violation calls RHS (line 12) to find a replaceable trajectory inside the robust neighborhood. To make a quick decision about the validation of the robust neighborhood, *RHS* checks the validity of the neighbors of *cp* at the edges of the neighborhood using the *cp*'s robustness degree to identify these edges (lines 1-6 in RHS). The RHS is a pure heuristic search for finding a valid trajectory inside the robust neighborhood with a minimum robustness of zero. Essentially, the heuristic search concentrates on quickly and effectively finding a valid trajectory as an immediate reaction against moving enemies. However, when there is no valid trajectory inside the robust neighborhood, RA* is recalled, generating a new path with its robust neighborhood considering all enemies that can be seen (line 14).

Algorithm 2 Functions needed for MTL-Robustness Monitoring

function loop()
1- X =sensor_results
2- **If** ρ=∅
3- ρ ←path generated by MTL_Robustness_Based_A*
4- **If** ρ=∅
5- Agent.Stop() // no path
6- **Else**
7- **If** sensor_results != ∅
8- X= X ∪ sensor_results
9- cp=violate_robustness(ρ,X,0, ϵ_{max})
10- **If** cp≠null
11- s_0= drone_position
12- ρ= RHS(s_0,cp, ρ, 0, ϵ_{max})
13- **If** ρ=null
14- ρ ←re-plan by RA*
15- Agent.move(ρ)

function violate_robustness(ρ,X, ϵ_{min}, ϵ_{max})
1- **for** pos ∈ ρ
2- **if** robustness_safety(pos, X,0, ϵ_{max})<0

```
3-       return pos
4-   return null

function RHS(s_0,cp, ρ, ϵ_min, ϵ_max)
1- neighbors = neighbors_of(cp, cp.r)
2- for each pos ∈ neighbors
3-   if robustness_safety(pos, X,0, ϵ_max)<0
4-       neighbors.remove(pos)
5-   if neighbors=null
6-     return null
7-   p= s_0
8-   open←p
9-   while open ≠ ∅
10-         p= pop(open)
11-         if(dist(p,q)==r)
12-            return construct_path(p)
13-      closed←p
14-      for n ∈ neighbors(p,0, ϵ_max)
15-       if n ∉ open
16-           n.h= robustness_liveness(n, q)-n.r
16-         open←( n)
17-  return null
```

By using MTL robustness to plan and monitor paths, the generated paths satisfy the initial mission constraints under all circumstances. However, when the drone has smaller visual range than the enemies, it might be discovered by one or more enemies before it sees them. This case can be dealt with as a special case by considering these areas as temporarily passable allowing the drone to escape the immediate risk towards the target. This obviously violates the safety property, but that is because of the physical capabilities of the drone and not related to the path planning and monitoring processes.

Figure 4 illustrates the approach using the Rassim simulator where the drone has unlimited time and fuel to reach its target (deadline=∞, fuel=∞, v =100, $\epsilon_{\min} = v$, $\epsilon_{\max} = v$):

$$\varphi = \Diamond_{[0,\infty]}\, q \wedge \Box_{[0,\infty]}\,(\neg \mathrm{WR}\,) \wedge \Box_{[0,\infty]}\big((\infty > f_{min} \wedge\ t < \infty) \Longrightarrow \neg VR \wedge \neg JR\big)$$

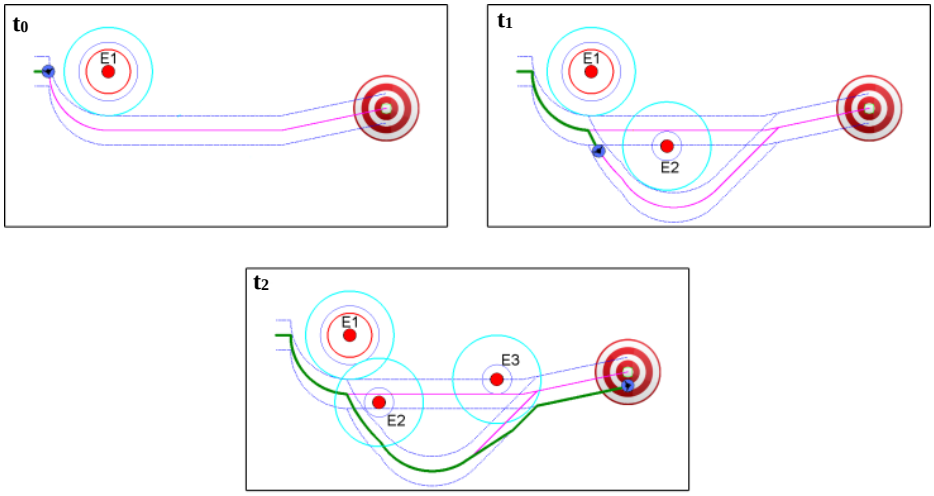

Figure 4: An example of RA* utilizes the robust neighborhood

Pink represents the planned path while green represents the path traversed by the drone to reach its current position. The robust neighborhood is shown in blue around the path. At time t_0, the drone detects enemy E_1 and updates its specification to become: $X = E_1$. *RA** finds a path that avoids enemies in X. At t_1, it sees E_2, which is a mobile enemy, and X becomes $\{E_1, E_2\}$. Since E_2 invalidates the whole robust neighborhood, *RA** is recalled to find another path with a new robust neighborhood. The drone sees E_3 and X becomes $\{E_1, E_2, E_3\}$, but part of the neighborhood is still valid. Here, *RHS* finds a valid trajectory to the target without doing replanning.

The case of limited resources is shown in Figure 5. Here, the mission specification is (deadline=$25ms$, fuel=$50g$, crt=0.1 g/ms, $f_{min} = 5g$, v =100, $\epsilon_{\min} = v$, $\epsilon_{\max} = v$):

$$\varphi = \Diamond_{[0,25]}\, q \wedge \Box_{[0,25]}\,(\neg \mathrm{WR}\,) \wedge \Box_{[0,25]}\big((\,f > 5 \wedge t < 25\,) \Longrightarrow \neg VR \wedge \neg JR\big)$$

Obviously, the drone cannot reach its target without accepting some risk. At t_0, the drone accepts some risk for the VR of enemy E_1. At t_1, it tries to avoid E_2, but it continues moving toward its path. Then, the drone evaluates the robustness of its current path. Since the deadline at t_1 is approaching, the path becomes robust despite the moving enemy and any further detected enemies. Therefore, the drone stops attempting to avoid E_2 and takes the direct path to the target without replanning when it saw enemy E_4.

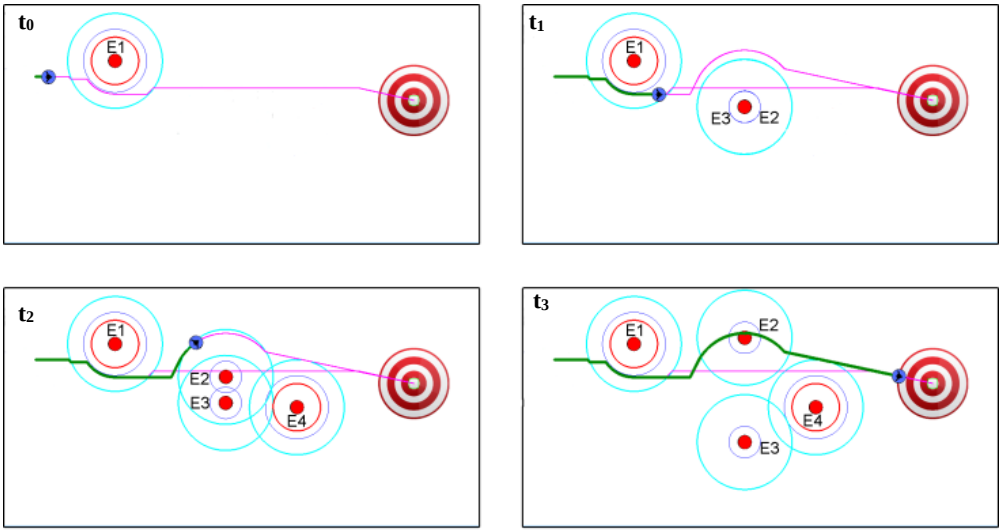

Figure 5: RA* path planning with limited resources

3.2.1.3 Evaluation

The main motivation of RA* [12] is to increase the possibility of mission completion in an adversarial world when the drone does not have sufficient resources to completely avoid all hazardous areas in presence of mobile enemies. Enhancing A* with MTL robustness helps to find a balance between risk avoidance and resource depletion. To show the effectiveness of the approach, we compare the average of planning time, travel time (i.e., path length), and the time spent inside risky areas of the A* algorithm and RA* in 100 randomly generated scenarios with static and moving enemies. We built a random scenario generator for Rassim to test our algorithm. The worlds are setup as 3000×1500 cell grids. The scenario generator randomly places the target, drone, and enemy assets using a normal distribution with multiple values of the standard deviation and mean using the world width and height. Killers are selected with a 60% probability, Jammers with a 20% probability, and Radars with a 20% probability. We initially set the min and max accepted robustness, $\epsilon_{\min}$ and $\epsilon_{\max}$, to the drone velocity. The experiments are conducted on a quad-core Intel i7 3.4GHz processor with 16GB RAM. We run the same scenarios with unlimited, limited (100 seconds), and insufficient (50 seconds) time. The results are shown in Table 1.

With unlimited resources, RA* successfully accomplishes the mission in all scenarios without accepting risk except in two cases, where enemies construct a virtual wall in front of the target. The drone was unable to reach its target without breaking through the VR and JR of enemies. With 50 seconds, RA* was unable to find feasible paths in 4% of the scenarios. The failed missions occurred because of the distribution of the enemies with WR. Since the drone must avoid WRs regardless of its resource condition, it ran out of time before reaching the target.

Table 1: Comparison between A* and RA* in seconds

	Avg. Planning Time	Avg. Travel Time	Avg. Accepted Risk	Mission Completion
A* (Unlimited Resources)	13.18	29.56	1.8676	98%
RA* (Unlimited Resources)	5.91	26.86	0.1022	100%
A* (Limited Resources)	10.36	26.71	1.227	90%
RA* (Limited Resources)	3.798	25.48	0.4948	100%
A* (Insufficient Resources)	7.129	25.01	0.6444	84%
RA* (Insufficient Resources)	2.499	23.43	0.509	96%

The average planning time of RA* in all cases is less than A*, because it prioritizes the drone safety by avoiding paths in narrow passages between enemies, while A* allows the drone to pass between enemies. Thus, A* sometimes traps the drone inside moving enemy ranges, consequently increasing replanning time. On the other hand, the RA* reduces replanning attempts by adjusting the trajectory of the current path from the robust neighborhood. Therefore, RA* generates safer, shorter, and faster paths overall than A* with and without limited resources.

With A*, the drone was unable to accomplish its mission in 10% and 16% of the scenarios with limited and insufficient resources, respectively. One possible reason for this occurrence is that when the drone is trapped inside a mobile enemy range for long time while trying to find an optimal path, it experiences resource depletion before reaching the target. Another reason may be that when the drone consumes its resources to completely avoid all enemies, it has no fuel or time left to accomplish the mission. In addition, the average path length (i.e., travel time) for paths generated by A* in both cases is surprisingly longer than the average length for RA* paths. In fact, A* attempts to find the shortest path, which makes its paths very tight around enemies. With mobile enemies, tight paths face replanning very frequently, increasing their overall lengths.

Although A* attempts to avoid all risky areas, the drone takes more risk with A* than with RA*, with and without limited resources. This situation is apparent when scenarios surround the drone with multiple enemies, trapping it. One scenario with moving enemies is shown in Figure 6. At time t_1 and t_2, A* spends significant time finding a valid path and stays in its position until enemies E_2 and E_3 move away from the optimal path. In similar scenarios, if enemies E_2 or E_3 move toward the drone, the drone will be trapped inside the risky area until the enemy moves away, which may cost the drone its life.

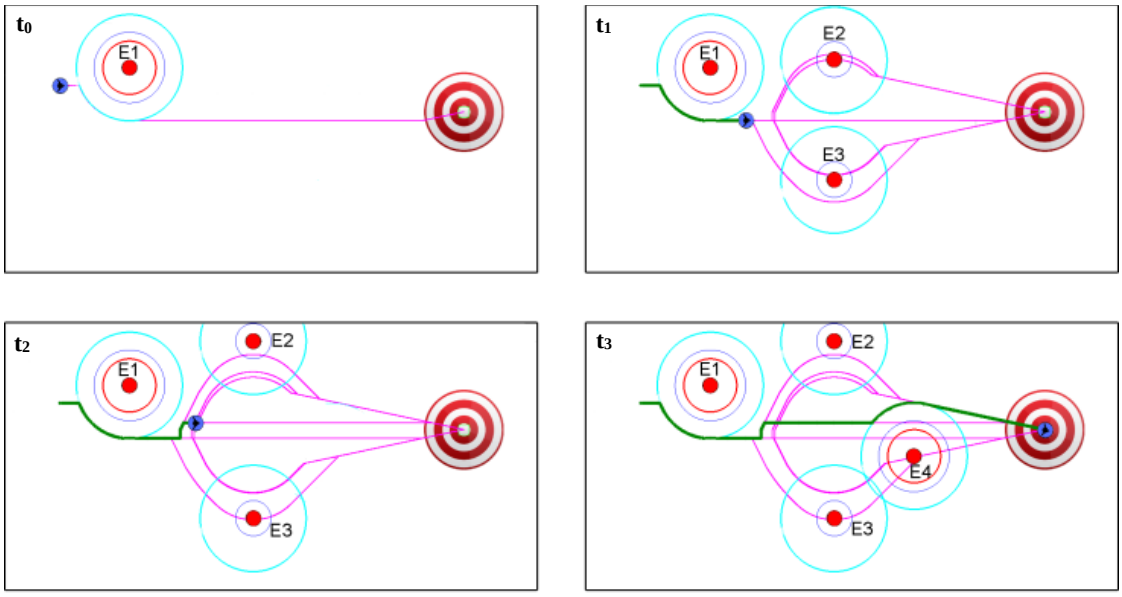

Figure 6: A*path planning in presence of mobile enemies

Figure 7 shows how RA* reacts to the same scenario in Figure 6 with unlimited resources. RA* can find a valid path without accepting any risk. The same scenario with limited resources has been shown in the previous section (Figure 5).

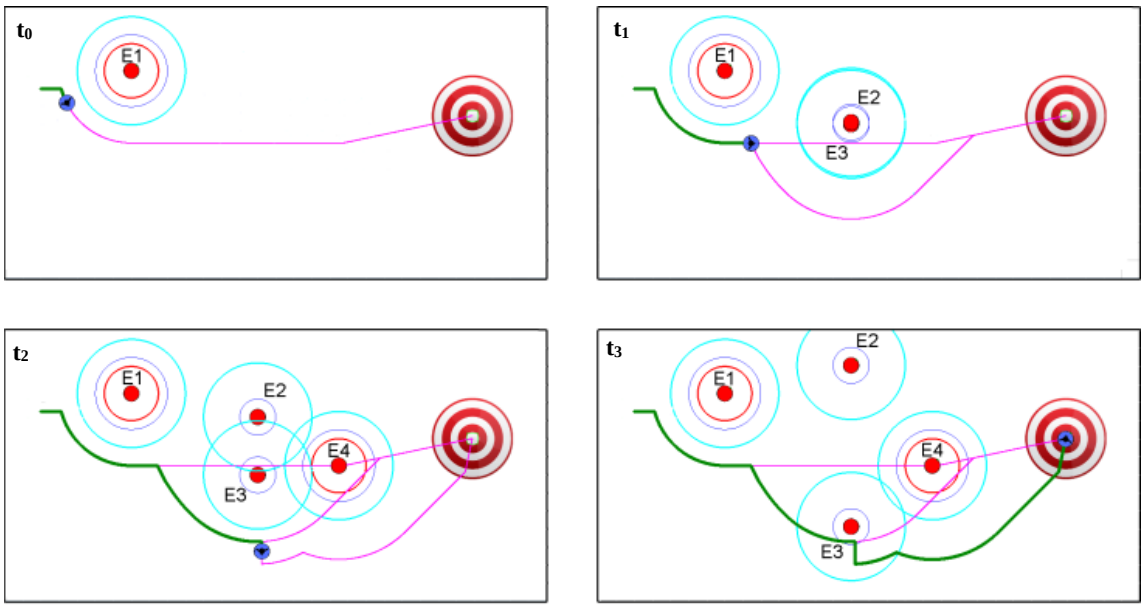

Figure 7: RA* plans with unlimited resources

3.2.2 Pattern Learning Module

In RA*, the robustness is computed based on the current observation of the enemy state without considering its future states assuming that the enemy is a static obstacle at each time step. This is impractical because it forces the drone to be reactive through re-planning whenever the sensor has detected a change. Realistically, the motion model of each mobile enemy is stochastic and needs to be carefully predicted through a well-designed prediction model. Such a prediction model can be utilized by the path planning algorithm to find more robust and stable paths. Hence, we extended our previous RA* algorithm to create Predictive-RA*, which enables the drone to behave in an anticipatory manner, weigh expected risk, and make decisions proactively rather than reactively where it might be too late to act with sufficiently low risk.

We developed a Pattern Learning Framework (PLF) for establishing relationships between the drone's state and the enemy's action pairs. By observing enemies in a state space, the PLF can identify behavioral models that explain the exhibited actions of different enemies. An offline learning technique infers these models using IRL, then the drone can utilize the PLF in an online setting to match the observed behavior of perceived enemies with the previously learned models. The assignment of these models helps drones equipped with the PLF to better anticipate the future actions of the observed enemies in the environment and safely plan its path to the target.

Our proposed framework PLF consists of two modules (Figure 8): the offline learning module and the online learning module. The offline learning module makes two assumptions about the observed enemies. The first assumption is that the enemy's behavior is Markovian, and therefore, makes its decisions based solely on its current conception of the world state. The second assumption is that enemies are purely-reactive. A purely-reactive agent makes its decisions based on the state of a different agent: its actions are assumed to be responding to the state of another agent. In our case, the enemy agent is assumed to be responding to the drone. Therefore, behavioral analysis is conducted with the assumption that exhibited behavior by the enemy is either a natural progression of its own state, or a response to a change in the observing drone's state. Based on the above assumptions, the world state for the enemy agent can be represented as the position and velocity of itself and the drone that is inside its detection range. The offline learning module is implemented using IRL to generate reward functions that attempt to explain the exhibited behavior given the trajectories produced by an enemy operating under a predefined behavioral model. This enemy agent is presumed to be acting optimally according to a particular policy and is modeled as a Markov Decision Process (MDP). Using the derived reward functions from the offline IRL module, the online learning module classifies newly observed enemies based on which reward function their behavior maximizes using an MDP to represent the enemy model and a KF to anticipate its next state.

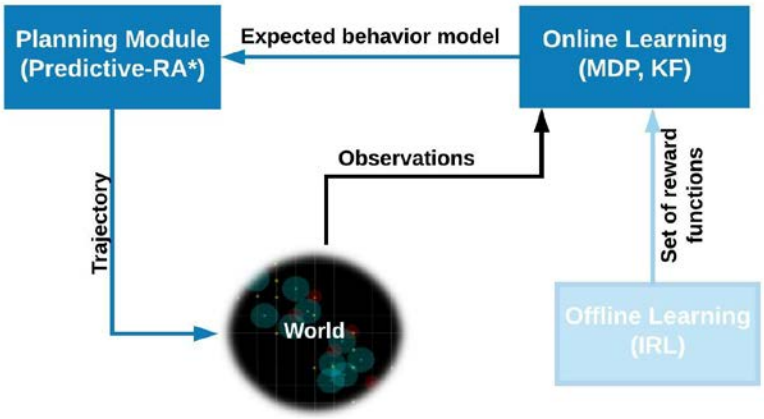

Figure 8: Pattern Learning Framework (PLF)

3.2.2.1 Markov Decision Process (MDP)

A Markov decision process (MDP) [13] is a formalization for modeling decision making. An MDP is represented by $M = (S, A, P, \gamma, r)$, where S is a set of states, A is a set of actions, P is a transition probability function for determining the subsequent state after taking a particular action in a particular state, γ is a discount factor that ranges from [0,1], and r is the reward function $r: S \times A \rightarrow R$. A policy $\pi: S \rightarrow A$ is a mapping from a state to an action. An optimal policy π^* is one such that $\sum_{t=0}^{N} \gamma^t r(s_t, a_t)$ is maximized, where the MDP has a finite horizon of N steps and $\pi^*(s_t) = a_t$.

The classical problem in MDPs assumes that all of the five elements are known and consists of finding an optimal policy that gives, for every state, the action that should be selected in order to maximize the reward function. For our purpose, we use an MDP to represent the enemy state such that its r is generated by the IRL during the offline learning phase and then used in the online learning setting to predict its future sequence of states.

3.2.2.2 Inverse Reinforcement Learning (IRL)

IRL [14] is a framework built on Markov Decision Processes (MDPs), where the goal of the apprentice agent is to find a reward function from expert demonstrations that could explain the expert's behavior [15]. In other words, the IRL problem is reverse-engineering a reward function that describes the behavior of an expert. In this project, an IRL agent has to determine a reward function given the trajectories an enemy has taken, assuming the enemy is acting optimally according to some MDP compliant policy.

In general, IRL occurs in a batch process where the dynamics, (i.e. transition probabilities function) P of the MDP, is unknown and where no interaction with the MDP is possible while learning. Thus, only transitions sampled from an MDP and without rewards (S, A, P, γ) are available to the learner. Most MDP-based IRL approaches assume that there is a set of m features associated with every state that fully determine the value of the reward function r. Since finding a general form solution for r is very difficult in this project, we assume it to be a linear combination

of a set of domain-specific features. Thus, for a given state *S*, the reward can then be expressed as the dot product $\phi_S.\alpha$ of a feature vector $\vec{\phi}_S = [\phi_1, \phi_2, .., \phi_m]$ and a weight vector $\vec{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_m]$. In this case, the IRL problem consists of estimating the values of the weight vector. The reader can refer to [14] for further details.

3.2.2.3 Kalman Filter (KF)

The Kalman Filter (KF) [16] was devised to allow multiple sources of uncertain information to be combined to reduce uncertainty. In principle, a KF works by assuming that the uncertainty can be represented using a Gaussian distribution (in as many dimensions as needed). By looking for the overlap between the individual distributions, the result is another Gaussian distribution with potentially a different mean and hopefully a smaller variance. In this project, we use the KF to model the enemy motion by predicting the changes to its position and velocity. After that, the sensor readings, i.e. observations, are combined with this prediction through the MDP to update the enemy state.

The KF operates under the following assumptions:

i) The Markov assumption
ii) System with linear dynamics
iii) Gaussian noise

Given the above assumptions, the KF can estimate the next enemy state (position and velocity) as a Gaussian distribution. The process the KF uses is documented below:

$$s_t^A = F_t s'^A_{t-\Delta t} + B_t \vec{u}_t \tag{16}$$

$$P_t = F_t P'_{t-\Delta t} F_t^T + Q_t \tag{17}$$

The above two equations are used for the prediction of the enemy state at time *t* based on observations from time $t - \Delta t$, which means Δt time has passed between the previous and current state. To account for a changing direction and magnitude of motion in (16), the control vector $\vec{u}_t$ and control matrix B_t are used. For this environment, the control vector $\vec{u}_t$ represents the acceleration of the enemy agent, which itself is dictated by its behavioral classification. The application of the acceleration vector will cause the enemy state to change. The control matrix B_t in equation (16) causes this change, and its exact composition is later defined.

The matrix P_t is the covariance matrix. The covariance matrix is a symmetric matrix that stores the variances of the variables on its diagonal, and the covariance between variables off of the diagonal:

$$P_t = \begin{bmatrix} \Sigma_{p,p} & \Sigma_{p,v} \\ \Sigma_{v,p} & \Sigma_{v,v} \end{bmatrix} \tag{18}$$

In equation (18), the form $\Sigma_{x,y}$ denotes the covariance between *x* and *y*. When the expression is in the form $\Sigma_{x,x}$ it denotes the variance of *x*. Exploiting a principle of matrices allows the values in the covariance matrix to be updated according to the system dynamics:

$$Cov(AX) = AXA^T \tag{19}$$

Substituting the prediction matrix F_t for the matrix A and the value of P_t at time $t - \Delta t$ for the matrix X, the first half of equation (17) is obtained. The second term Q_t in (17) is a covariance matrix that accounts for the process noise, which for our purposes helps to account for the change in velocity the enemy might take during the Δt time interval.

The Kalman gain function determines how much to change from the prediction of the state at time t to the observation at time t. The gain function is defined as:

$$K' = P_t H_t^T (H_t P_t H_t^T + R_t)^{-1} \tag{20}$$

The matrix H_t is responsible for converting from the state space to the measurement space. In the discussed state space, the state space and measurement space are identical (position and velocity pairs), which means the H_t matrix will be an identity matrix for the purposes of this paper. The matrix R_t is the noise of the sensor reading. This diagonal matrix contains the variances of normal distributions for the corresponding dimensions. The value of $H_t P_t H_t^T$ represents the expected sensor readings for the covariance matrix P_t using the identity in equation (19). The gain function is used to obtain a better approximation of the current state and covariance of the state:

$$s'^A_t = s^A_t + K'(\vec{z}_t - H_t s^A_t) \tag{21}$$

$$P'_t = P_t - K' H_t P_t \tag{22}$$

In equation (21), the value of the vector $\vec{z}_t$ is the sensor reading at time t, which in other words is the observed state. The vector s^A_t signifies the updated estimate of s^A_t when considering the sensor readings. The same principle applies to P'_t. The value of $H_t s^A_t$ represents the expected sensor readings given the estimated state s^A_t. Subtracting this product from the actual observed sensor readings $\vec{z}_t$ provides the difference between the predicted and observed states. The gain function K' is applied to this difference (which can be considered as the error) and the error is added to s^A_t to yield s'^A_t, which is the updated estimate of s'^A_t. The general idea is also applied in equation (22), but with regards to the covariances. Lastly, the updated estimates (s'^A_t and P'_t as opposed to $s^A_t$and P_t) are used as the values for equation (16) and equation (17) the next time the KF is used. This portion of the predictive process helps to fine-tune the KF so the difference between the predicted and observed states becomes smaller with time, which increases the accuracy of the prediction model.

3.2.2.4 Pattern Learning Framework (PLF)

Given the assumptions that the enemy makes decisions based only on its current state (Markovian) and on the ally drone's current state (purely reactive), the world state for the enemy can be represented as:

$$\vec{s} = \left[x_{drone}, y_{drone}, \dot{x}_{drone}, \dot{y}_{drone}, x_{enemy}, y_{enemy}, \dot{x}_{enemy}, \dot{y}_{enemy}\right]^T$$

where, for example, (x_{drone}, y_{drone}) and $(\dot{x}_{drone}, \dot{y}_{drone})$ denote the position and velocity of the drone (PLF agent) respectively. On the other hand, (x_{enemy}, y_{enemy}) and $(\dot{x}_{enemy}, \dot{y}_{enemy})$ indicate the position and velocity of the enemy. Actions in this problem space are in the form of

$\vec{u} = [\ddot{x}, \ddot{y}]^T$,which represents an acceleration vector (i.e. action vector in equation (16)) taken by the enemy.

Given the earlier claims, the problem domain has a set of potential states $S \subseteq \mathbb{R}^8, A \subseteq \mathbb{R}^2$ because the state vector $\vec{s}$ has a dimensionality of 8 and the action vector $\vec{u}$ has a dimensionality of 2. The high dimensionality of the state space would make the learning process intractable. To mitigate this issue, a feature transformation function Φ was developed to simplify the state space and make it possible to use for performing learning. This reduced state space is called the feature space. We found that predicting a part of the state that is dependent on the drone's actions gives a better reward structure. This is in line with the work of Pathak et al. [18]. For the PLF, the feature space includes two types of quantities. First, quantities that are state-space homogeneous, where values in the state space are based on distance rather than absolute position. Second, quantities that are measured relative to each other, such as the enemy's velocity relative to the drone's position. Rather than raw state space vectors $\vec{s}$, feature vectors $\vec{\phi}$ with these characteristics generalize better across situations with drastically different state space positions, but fundamentally describe similar instances.

Determining the acceleration vector taken by the enemy at each time step is the core process of the behavior identification problem during offline learning. Different acceleration vectors under identical state conditions, or rather identical feature states, is what differentiates behavior types. For instance, when the enemy maintains an acceleration vector $\vec{u}$ = [0, 0] throughout, it is simply traveling in a straight line with an unchanging velocity and its future state can then be optimally predicted by using only the KF. The system becomes nonlinear given control inputs in the form of acceleration vectors and to accurately predict the future state of the enemy, the KF must be provided with the expected acceleration vector $\vec{u}$ for the enemy. These vectors are generated using the MDP and the reward functions learned from the offline learning module (Figure 8).

3.2.2.4.1 Offline Learning using IRL

The offline learning module aims to derive reward functions that describe the behavior of enemies by following the same IRL approach implemented in Ng and Russel [14]. Obtaining an approximation of the reward functions provides the likely actions that enemies with the same behavior type will exhibit in online environments. The offline learning works in three steps: (1) generating state-action pairs, (2) selecting the feature set, and (3) optimizing the selected features.

1) State-Action Pair Generation

The learning process requires a data set that provides the sequence of state-action pairs:

$$\text{Log Data} = \{\vec{s}_1 \xrightarrow{\vec{u}_1} \vec{s}_2 \xrightarrow{\vec{u}_2} \vec{s}_3 .. \xrightarrow{\vec{u}_{n-1}} \vec{s}_N\} \quad (23)$$

However, logged data is often not contiguous from start to finish. In other words, the drone may observe parts of an enemy's trajectory, and may observe that enemy (or different enemies) at different points in time with moments in between where no observation takes place. The log data then becomes temporally discontinuous and fragmented. Furthermore, since the learning occurs based on how the enemy behaves within the radius of observation, the learning algorithm cannot make any assumptions about the enemy behavior once the enemy is outside that radius. The

learning is with respect to the reactive behavior of enemy agents rather than their global unobserved behavior, which means inference between fragmented observation sections is inadvisable. Therefore, rather than a complete trajectory from start to finish, a set of smaller trajectories are used, splitting the logged data into multiple incomplete, but individually continuous trajectories:

$$\text{Log Data} = \left\{ \begin{array}{c} \left\{ \vec{s}_{1,1} \xrightarrow{\vec{u}_{1,1}} \vec{s}_{1,2} \xrightarrow{\vec{u}_{1,2}} \vec{s}_{1,3} \dots \xrightarrow{\vec{u}_{1,m-1}} \vec{s}_{1,m} \right\}, \\ \left\{ \vec{s}_{2,1} \xrightarrow{\vec{u}_{2,1}} \vec{s}_{2,2} \dots \xrightarrow{\vec{u}_{2,n-1}} \vec{s}_{2,n} \right\}, \dots, \left\{ \vec{s}_{k,1} \xrightarrow{\vec{u}_{k,1}} \vec{s}_{k,2} \dots \xrightarrow{\vec{u}_{k,l-1}} \vec{s}_{k,l} \right\} \end{array} \right\} \tag{24}$$

Note that the above strategy generalizes to a set containing one complete trajectory (akin to the format in equation (23)), but equation (24) is shown as well for the sake of generality. The set of these individually contiguous trajectories is the data the IRL algorithm uses to generate the reward function for the behavior type that produced those trajectories. Offline learning occurs with the assumption that all observed enemies are of a common type, which means every observed enemy in the offline learning process is assumed to be exhibiting the same behavior model.

The logged data is the path the enemy has taken through the environment, a sequence of steps where an action $\vec{u}_t$ transforms state $\vec{s}_t$ into $\vec{s}_{t+1}$. That process of 'stepping' forward to the next state takes the following form:

$$step(\vec{s}_t, \vec{u}_t) \rightarrow \vec{s}_{t+1}^{\text{enemy}} \tag{25}$$

The $\vec{s}_{t+1}$ has an "enemy" superscript to denote that the step function only yields the enemy state portion of $\vec{s}_{t+1}$, because the drone state progression is independently determined by the drone's current planned trajectory. The action vector $\vec{u}_t$ is not given, but rather inferred from the pair of states that would surround the hypothetical action vector. This action vector is determined from equation (25) and a null action $\vec{0}$, which is an action vector composed entirely of 0:

$$\vec{u}_t = \vec{s}_{t+1} - step(\vec{s}_t, \vec{0}) \tag{26}$$

The action vector is then asserted to be some function of the state residual between the observed subsequent state $\vec{s}_{t+1}$ and the subsequent state that would have occurred if no action was taken, which is provided by $step(\vec{s}_t, \vec{0})$. The output of the $step(\vec{s}_t, \vec{u}_t)$ function is described in more detail in equation (32), but suffice to say that the function uses a KF with the action vector $\vec{u}_t$ as the control input.

2) Feature Selection

The selection of features is a critical problem that is highly domain-specific. It dictates the structure and performance of the reward function, especially in high-dimensional state spaces with ostensibly unrelated state variables. Selecting inappropriate features results in useless reward functions, and reasonably chosen ones feed the learning algorithm with more domain-specific knowledge.

In this project, we develop a feature transformation function that takes the form of $\Phi: S \times A \rightarrow \vec{\phi}$, where $\vec{\phi}$ is the vector of features, S is the set of states, and A is the set of actions. The feature function requires a current state and the action that immediately followed, so calls to

the feature transformation function would appear as $\Phi(\overrightarrow{s_t}, \overrightarrow{u_t})$, where $\overrightarrow{s_t}$ is the state at time *t* and $\overrightarrow{u_t}$ is the action taken at time *t*. For our problem domain, features were chosen with respect to the assumption that the enemy behavior is Markovian and purely reactive. The features, therefore, relate the velocity of the enemy with respect to itself and the current state of the drone. To adequately compare these potentially unrelated values, a standardization technique was devised. The feature standardization technique is illustrated in Figure 9. Figure 9.A shows an example of the state space, where the enemy is traveling northeast, and the drone is traveling southwest, such that the drone is approximately directly above the enemy.

Consider three vectors: one pointed in a direction parallel to the heading of the enemy (Figure 9.B, vector v_0), one in a direction parallel to the allied drone (Figure 9.C, vector v_0), and one pointed toward the position of the allied drone from the position of the enemy (Figure 9.D, vector v_0). Each of these vectors is hereafter referred to as the 0-indexed vector v_0 for their given set of *K* vectors. The remainder of the vectors in each of the sets are indexed clockwise, where v_n is $\frac{n}{k}2\pi$ radians from the direction of the v_0 vector of that set. Given the three previously defined v_0 vectors, this results in a total of *3K* vectors. Larger values of *K* correspond to more accuracy, but lead to more computation during the reward function generation process and during the online evaluation of behavior. The value *K* is set to *5* in the diagram to minimize clutter.

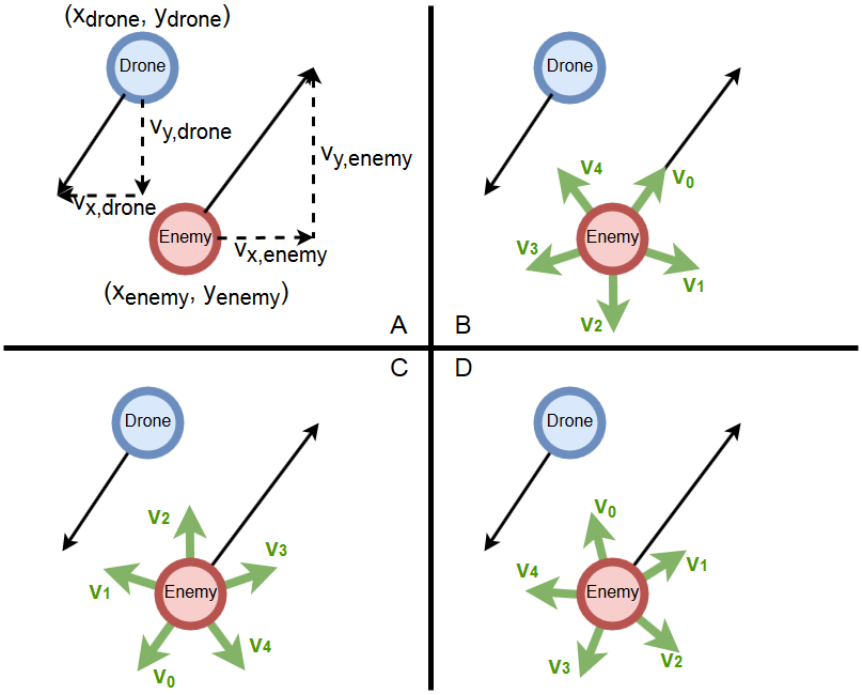

Figure 9: State space and feature space

The features are then computed as the similarity between the enemy velocity vector after applying the action $\overrightarrow{u_t}$ and each of the vectors within each set. Each individual feature ranges from [0,1], where a value of 0 corresponds to the two vectors pointing in exact opposite directions. Each of the vectors in each set are scaled to the maximum speed of the enemy.

3) Optimization

Once appropriate features have been selected, the offline learning problem becomes learning the following vector:

$$\vec{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_{3K}]^T \quad (27)$$

This vector represents the weight attributed to each feature. Note that the size of the weight vector $\vec{\alpha}$ is three times the value of *K* discussed in the previous *Feature Selection* section, to represent the three sets of *K* vectors.

The reward function is a linear combination of the features, with coefficients for each of the features dictated by the weight vector $\vec{\alpha}$ in equation (27). The reward function, therefore, takes the following form:

$$rwd(\vec{\alpha}, \vec{\phi}) = \sum_{i=1}^{3K} \alpha_i \phi_i \quad (28)$$

The optimization procedure above must be run for any given behavior type that could be observed in the environment. Running this procedure for the various behavior types produces a set Λ of $\vec{\alpha}$ vectors, each representing the reward function for the behavior type that was processed. This set is used in the online observation to classify encountered enemies and determine the most likely actions they will take.

3.2.2.4.2 Online Learning using IRL

The online learning module observes newly detected enemies and classifies their behaviors based on the reward functions they maximize. The set of reward functions, Λ, generated by IRL, is maximized by the MDP to assign the appropriate behavioral model to the enemy based on its online observed states. Then, the KF is used to predict the future states of the enemy by processing the action vector $\vec{u}$ assigned by the MDP during the classification phase.

1) Representation of Behaviors

As mentioned above, the behaviors that are learned offline are stored in the set Λ. This set contains different $\vec{\alpha}$ vectors for the different features' weights. Using these vectors with the reward function template defined in equation (28) allows for modeling different behaviors with distinct reward functions. The action $\vec{u}$ that a particular behavior type will execute given the current state $\vec{s}$, is the action $\vec{u}^*$ that maximizes the reward function associated with that behavior type.

The subsequent state that the enemy is believed to proceed to is determined by the *step* function, originally described in equation (25). The *step* function takes a state and an action then produces the state that would result from applying the action in the given state. The function uses a KF to predict how the state will be transformed with the provided control input. For our particular domain, the KF employs the following prediction and control matrices:

$$F_{\Delta t} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

$$B_{\Delta t} = \begin{bmatrix} (\Delta t)^2/2 & 0 \\ 0 & (\Delta t)^2/2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \tag{30}$$

The KF is the main mechanism that dictates the return state of the *step* function. Given the dynamics defined by the matrices in equations (29, 30), along with the KF predict step defined as [17]:

$$\vec{s}_t = F_t \vec{s}_{t-\Delta t} + B_t \vec{u}_t \tag{31}$$

the *step* function then assumes the following form:

$$step(\vec{s}_t, \vec{u}_t) = F_{\Delta t} \vec{s}_t^{\,\text{enemy}} + B_{\Delta t} \vec{u}_t \tag{32}$$

While the *step* method is a mechanism for progressing the MDP that represents the enemy state one time-step forward, it also acts as a tool for simulating the motion dynamics of the environment through the KF.

Note that the KF expects a state vector with a dimensionality of four because it is used only for predicting the enemy state. Since the drone already has a previously articulated and known plan, the KF does not need to predict its future state. The world state for an observed enemy is represented as the union of two state vectors: the observing drone's state and the observed enemy's state. Determining how this vector changes in the future is done based on the drone's plan along with the forecast of the enemy's state, which in turn is based on the anticipated state of the drone in the preceding time step. This becomes a pure prediction problem once the enemy has been classified with a behavior type.

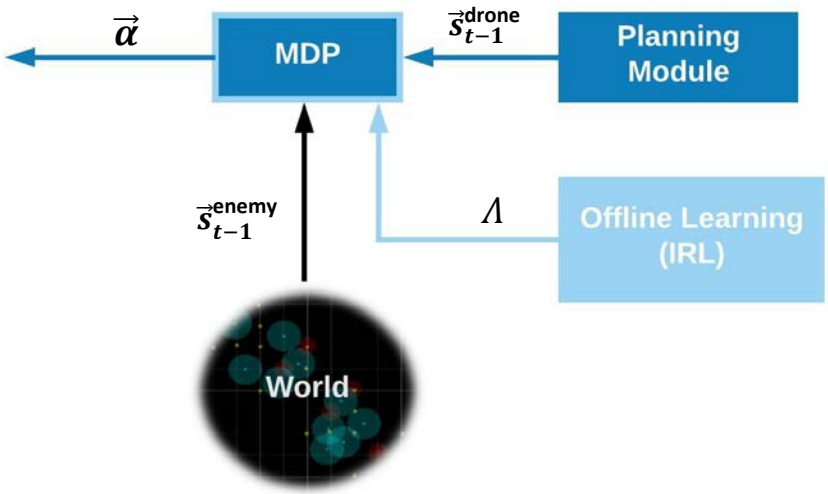

Figure 10: Behavior classification process

2) Classification of Newly Discovered Enemy

Before any prediction can be determined, a behavioral model must first be assigned to that enemy. This process involves extracting the actions an enemy has executed under the given state conditions. A similar procedure described in equation (26) is employed for this purpose. Given that $\vec{s}_t$ is the currently observed state and the enemy was observed the timestep before, $\vec{u}_{t-1} = \vec{s}_t - step(\vec{s}_{t-1}, \vec{0})$ is used to find the residual between the $\vec{s}_t$ that was observed and the hypothetical $\vec{s}_t$ that would have occurred if an action vector $\vec{0}$ was applied. This action $\vec{u}_{t-1}$ along with the preceding state $\vec{s}_{t-1}$ are used as arguments for each of the behavior types stored in the set Λ, and whichever reward function is maximized under the exhibited state-action pair is the most likely behavior the enemy is exhibiting (Figure 10).

Although the above procedure is described in the context of discovering a not previously encountered enemy in the environment, the same process applies when the enemy changed its currently assigned behavior type. In this case, a different $\vec{\alpha}$ should result in a larger reward for an observed action and the assigned behavioral model would shift to the new type. This procedure is formalized in **Algorithm 3** below:

Algorithm 3: assign_behavior ($\Lambda, \vec{s}_{t-1}, \vec{s}_t$)

1- $\vec{s}_t^{\text{null}} = \boldsymbol{step}(\vec{s}_{t-1}, \vec{0})$
2- $\vec{u}_{t-1} = \vec{s}_t^{\text{null}} - \vec{s}_t$ (26)
3- $rwd_{max} = 0, \vec{\alpha}_{max} = \vec{0}$
4- **for** $\vec{\alpha}_i \in \Lambda$:
5- $rwd_i = rwd(\vec{\alpha}_i, \boldsymbol{\Phi}(\vec{s}_{t-1}, \vec{u}_{t-1}))$ (28)
6- **if** $rwd_i > rwd_{max}$:
7- $rwd_{max} = rwd_i, \vec{\alpha}_{max} = \vec{\alpha}_i$
8- **return** $\vec{\alpha}_i$

3) Prediction of Enemy with Classified Behavior

Predicting the enemy's behavior once a behavioral model has been applied becomes a task of successive **step** calls. The process uses the current world state and builds a new future state by iteratively producing the sequence of states that could occur. The hypothetical sequence of states is contingent on the enemy's assigned behavioral model and the drone's current plan.

For example, suppose we want to predict the world state at $\vec{s}_{t+1}$. The drone's portion of this state is assumed to be predetermined from its plan. On the other hand, the enemy's portion of the state is dictated by the action $\vec{u}_t$ that is derived from its designated behavior type and the world state $\vec{s}_t$ immediately preceding that action (Figure 11).

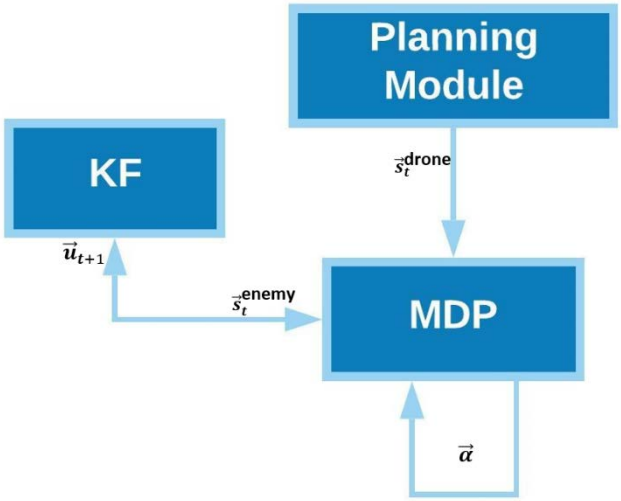

Figure 11: Prediction process for enemy behavior

The principle generalizes to determining the state at $t+i$. If only the enemy state at time $t+i$ is needed, the drone's plan needs to only be known up to the time step $t+i-1$. Otherwise, if the whole estimate for the world state $\vec{s}_{t+i}$ is desired, the drone's plan must extend one more step to $t+i$. The algorithm for this procedure is formalized in **Algorithm 4**, where i is the number of time steps ahead to forecast, $\vec{\alpha}$ is the current behavioral model, $\vec{s}_t^{\text{enemy}}$ is the currently observed enemy state, and the sequence of planned drone's states B, whose first element is the current drone state at time t.

Algorithm 4: $\textbf{forecast}\left(i, \vec{\alpha}, \vec{s}_t^{\text{enemy}}, B\right)$

1- $\overrightarrow{curr} = \vec{s}_t^{enemy} \cup B_t$
2- $\boldsymbol{for}\ j\ \textbf{from}\ t+1\ \textbf{to}\ t+i$:
3- $\quad \vec{u}_j^* = arg\ \max_{\vec{u}} \left(rwd\left(\vec{\alpha}, \boldsymbol{\Phi}(\overrightarrow{curr}, \vec{u})\right)\right)$
4- $\quad \overrightarrow{curr} = \boldsymbol{step}\left(\overrightarrow{curr}^{\text{enemy}}, \vec{u}_j^*\right) \cup B_j$
5- $\boldsymbol{return}\ \overrightarrow{curr}$

3.2.2.4.3 Predictive-RA*

In order to plan for robust paths in hostile environments, the dynamics of the adversaries need to be embedded within the path planning algorithm. To increase the robustness of the drone's path, the predicted states of the enemy must be considered in the robustness computation. Any prediction model cannot be precise enough to give one exact position for the enemy, but instead it answers the question of how likely the enemy is going to be in a specific area in the world state. Hence, we extend the MTL specification of our drone in equation 3 to include the following predicate:

$$\square_{[0,\text{deadline}]}\ \left(\wedge_{t=0}^{v} P\left(\text{inside}\left(\bullet_{t|0}\ WR\right) < \alpha\right)\right) \tag{33}$$

The interpretation of this predicate is that during the mission time, the probability of the drone being inside the predicted WR of the enemy set must be less than a threshold of α within the next v time. The threshold can be scaled according to the problem domain and the importance of the drone safety. The finite horizon v is imposed to restrict the robustness monitoring up to the limited vision range of the drone since resolution for potential violation in the far future is likely to only waste computation resource and time. However, this predicate returns a Boolean value while the robustness of equation (14) returns a numerical value. In order to resolve this conflict, we simply compute the robustness of equation 33 as follows:

$$\begin{aligned} P\left(\text{inside}\left(\bullet_{t|0}\ WR\right) < \alpha\right) &\Rightarrow \left(1 - P\left(\text{inside}\left(\bullet_{t|0}\ WR\right)\right)\right) \times PR \\ P\left(\text{inside}\left(\bullet_{t|0}\ WR\right) \geq \alpha\right) &\Rightarrow -\infty \end{aligned} \tag{34}$$

where PR is the preferred robustness value. Equation 34 indicates that the position with a probability less than α is considered robust and its robustness value depends on its probability of being outside the WR multiplied by the preferred robustness value (i.e. weight). On the other hand, the robustness of the position with the risk probability higher than α is set to be negative infinity to exclude these positions from the search space for Predictive-RA*.

The drone specification in equation 3 becomes:

$$\varphi = \Diamond_{[0,\text{deadline}]}\ q \wedge \square_{[0,\text{deadline}]}\left(\neg WR\right) \wedge \square_{[0,\text{deadline}]}\left(\wedge_{t=0}^{v} P\left(\text{inside}\left(\bullet_{t|0}\ WR\right) < \alpha\right)\right) \tag{35}$$

The probability of the drone being *inside* the WR cannot be computed directly from the Gaussian distribution given by KF. The distance between the drone's path points and the distribution of the enemy state must be considered. To compute that, we generate a new non-central χ^2 distribution [18] using the Gaussian distribution of the enemy state.

Let (x,y) be the drone's known position and the enemy position (μ_{x_e}, μ_{y_e}) is the assumed position by a Gaussian distribution as $\sim\mathcal{N}(\mu, \sigma)$ from KF. The distance between the drone and the enemy is computed as:

$$r^2 = \left(x - \mu_{x_e}\right)^2 + \left(y - \mu_{y_e}\right)^2 \tag{36}$$

We can compute the non-centrality parameter of the non-central χ^2 distribution as follows:

$$\lambda = r \tag{37}$$

Then, the probability density function for the χ^2 with k=2 degrees of freedom is given by:

$$f(r, k) = \frac{1}{2}\left(e^{-r}\right) I_{k/2-1}(r) \tag{38}$$

where $I_k(r)$ is the modified Bessel function of the first kind of order with k degrees of freedom. The cumulative distribution function (cdf) can be used to find the likelihood that the distance between the drone and the enemy set is less than the WR (i.e. the drone is inside WR):

$$\mathrm{P}(\mathrm{inside}(\mathrm{WR})) = \int_0^{\mathrm{WR}} \mathrm{f}(\mathrm{r},\mathrm{k})\ \mathrm{dr} \tag{39}$$

The Predictive-RA* algorithm is shown in Algorithm 1. It is built on A*, but with essential modifications. First, the MTL robustness computation of the safety property (line 8 in neighbors function) is included within the cost function g to favor robust trajectories. The positions that are expected to be inside the WR with probability higher than α would be excluded from the search space (line 6), preventing the drone from taking paths through them. Positions with less risk are considered passable. Their robustness degree depends on how far they are expected to be from the WR in equation 34. In other words, trajectories with larger distances away from the enemy set are dominated in Predictive-RA*. Subtracting the expected robustness from the g function in line 9 creates uphill terrains around the locations where the enemy is high likely to be there in the near future. On the other hand, locations far away from these areas are dealt with as downhill terrains to encourage the drone to pass through them. However, the robustness degree is constrained by the resource limitations (line 6). Thus, the trajectory robustness is maximized based on the available fuel and time for the drone to accomplish its mission (conditions in equations 13-2,13-3). The last modification is using KF in equations 1,2,8 and 9 to predict the states of the enemy set in the world space over the planning time (lines 12 and13 of Predictive-RA* and lines 3 and 5 in neighbor function).

RA* considers the visual range (VR) of the enemies as risky as their weapon range. However, by forcing the drone to avoid being seen by enemies, RA* increases the resource consumption. Moreover, RA* works properly under the assumption that the drone's VR is always larger than the maximum VR of the enemy set. Realistically, the drone cannot avoid being seen in environments with capable enemies. In order to bridge this gap, the *trajectory monitor* evaluates the robustness of the MTL safety property in equation 16 only when the drone becomes inside VR of one or more enemies. In other words, the decision of re-planning a new robust path for the drone depends on the answer to the question of "*whenever the probability of the drone passing inside the VR of the enemy set within the next v time is at least β, is the current trajectory still robust?*" This question can be written as an MTL constraint:

$$\left(\bigwedge_{t=0}^{v} \mathrm{P}\left(\mathrm{inside}\left(\bullet_{t|0}\ \mathrm{VR}\right)\right) \geq \beta\right) \Rightarrow \Box\ \mathrm{P}\left(\mathrm{inside}\left(\bullet_{t|0}\ \mathrm{WR}\right) < \alpha\right) \tag{40}$$

Algorithm 5 Predictive-RA*

Function Predictive_RA*(s_0,q)

```
1-  closed = ∅
```

```
2-  s_0.g= ¬WR(p_0, E_0)
3-  s_0.h= Euclidean(s_0, q)
4-  s_0.f = s.g+s.h
5-  open.insert(s_0)
6-  while open ≠ ∅
7-    current = pop(open_queue)
8-    if current = q
9-      return construct_path(q)
10-   open.remove(current)
11-   closed.insert(current)
12    x̂'_t = ∪_{i=0}^{n} x̂'_{i,t}  ∀ e_i ∈ E    (21)
13-   P'_t = ∪_{i=0}^{n} P'_{i,t}  ∀ e_i ∈ E    (22)
14-    for neighbor ∈ neighbors(current,X, x̂'_t, P'_t)
15-      if neighbor ∉ closed
16-        if neighbor ∉ open
17-          open.insert(neighbor)
18-        else
19-          openneighbor = neighbor ∈ open
20-          if neighbor.g > openneighbor.g
21-            open.remove(openneighbor)
22-            open.insert(neighbor)
23-  return false // no path exists
Function neighbors(p,X_i, x̂'_t, P'_t)
1-  neighbors = neighbors_of(p,1)
2-  for n ∈ neighbors
3-    x̂_{t+Δt|t} = ∪_{i=0}^{n} x̂_{i,t+Δt|t}  ∀ e_i ∈ E   (16)
4-    P_{t+Δt|t} = ∪_{i=0}^{n} P_{i,t+Δt|t}  ∀ e_i ∈ E   (17)
5-   if   P(inside(•_{t|0} WR) < α)      (33)
6-     if(fuel-fuel_to_n>0 & deadline-time_to_n>0)
7-      n.r=(1 − P(inside(WR_n))) × PR
8-      n.g = d(n,p)- n.r
9-      n.h= Euclidean(neighbor,q)
10-     n.f = n.g + n.h
11-     n.parent = p
12-  return neighbors
```

The developed monitoring algorithm is shown in Algorithm 6. The loop moves the drone on the path and monitors robustness simultaneously. The drone's sensor data (line 2) is used to update the KF's predicted values (lines 3 and 4) whenever it sees the enemies. Line 5 checks whether the current path is expected to lead the drone into the VR of the enemy in equation 21. When the probability of this action is above a certain threshold β, a robustness evaluation of the mission constraint is conducted in the *violate_robustness* function (line 6 of the loop). The robustness is computed using the drones' path points, the state predictions, and the safety property (equation 34). Once a violation is anticipated (line 4 in *violate_robustness*), Predictive-RA* is executed again, generating a new robust path considering all enemies that can be seen and predicted (line 10 of the loop).

Algorithm 6 Trajectory Monitoring

```
function loop()
1- ρ: current path
2-  z⃗_t= sensor_data
3-   x̂'_t = ∪_{i=0}^{n} x̂'_{i,t}  ∀ e_i ∈ E        (21)
4-   P'_t = ∪_{i=0}^{n} P'_{i,t}  ∀ e_i ∈ E        (22)
5-   if(∧_{t=0}^{v} P (inside(•_{t|0} VR)) ≥ β )     (40)
6-          if violate_robustness(ρ, x̂'_t, P'_t)
7-                ρ ←re-plan by Predictive-RA*
8-       drone.move(ρ)
```

```
function violate_robustness(ρ, 𝔛′_t, P′_t)
1-    for pos_i ∈ ρ
2     𝔛_{t+Δt|t} = ⋃_{i=0}^{n} 𝔛_{i,t+Δt|t}  ∀ e_i ∈ E   (1)
3-    P_{t+Δt|t} = ⋃_{i=0}^{n} P_{i,t+Δt|t}  ∀ e_i ∈ E   (2)
4-       if (⋀_{t=0}^{v} P(inside( •_{t|0} WR) ≥ α))        (40)
5-          return true
6-    return false
```

3.2.2.5 Evaluation

The testing environment is a continuous two-dimensional map with various enemies, and a randomly navigating drone. The experiment aims to test the classification accuracy and the predictive ability of the method when different amounts of data are used to learn. For testing the algorithm, three different behavior types were specified for the described problem domain: a *chasing* behavior, an *interception* behavior, and an *orbiting* behavior. Agents exhibiting the *chase* behavior always travel directly to the drone. The *interception* behavior accounts for the current heading of the drone and attempts to move towards an intercept point a certain distance ahead of the drone. The *orbiting* behavior simply circles around a particular point, "guarding" a certain region.

We tested our agents using RASSim ATE3, which is a simulator produced by the Air Force Research Lab to simulate ally and adversarial drones. Data for offline learning was produced running three separate simulations where the drone wandered in an environment populated by enemies with a single behavior type. Each simulation was run long enough to accrue a large number of sample trajectories observing how each behavior type makes decisions under various different state conditions. For the online portion of testing the drone was again tasked with randomly wandering different environments, but rather than simply observing, the drone used the set of learned behavior models Λ to avoid the enemies.

To evaluate the strength of the learning process, each trial was run on different amounts of learned data – in other words, identical world conditions (trials) were run multiple times, but each time a subset of the learned data from the offline portion was used. For our experimentation, we used 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, and 1000 data points for each trial, where a data point is a state-action pair. A trial is a random arrangement and behavioral assignment of enemies. Furthermore, we ran an identical process where the behavior types were entirely homogenous throughout the environment, to evaluate the improvement in classification accuracy for each behavior type individually.

The test results indicate that the algorithm successfully predicts and classifies given more input data, and performs reasonably well, even with small amounts of training data. However, a discrepancy emerges due to the similarity between the '*chase*' and '*intercept*' behaviors. The *chase* and *intercept* behaviors become nearly indistinguishable once the enemy is behind the drone agent and the two are moving in the same heading (a situation that becomes inevitable with both behaviors when the drone is faster). This situation results in the algorithm placing similar weights for both *chase* and *intercept*, which increases the amount of misclassification for these behavior types.

The data shows in Figure 12 a decrease in the prediction error; the distance between the predicted location of the enemy 10 seconds after observation and its actual location. Moreover, the data in Figure 13 shows an increase in the classification accuracy; the percentage of observed enemies correctly classified with one of the three behavior types; when more input data was used to generate the reward functions. *Intercept* behavior seems to suffer from the above dilemma the most with respect to the classification accuracy, although all behavior types improve their predictive qualities with more training data. Furthermore, the prediction results are also affected by the misclassification, because often a misclassification can result in an erroneous prediction when forecasting further in advance.

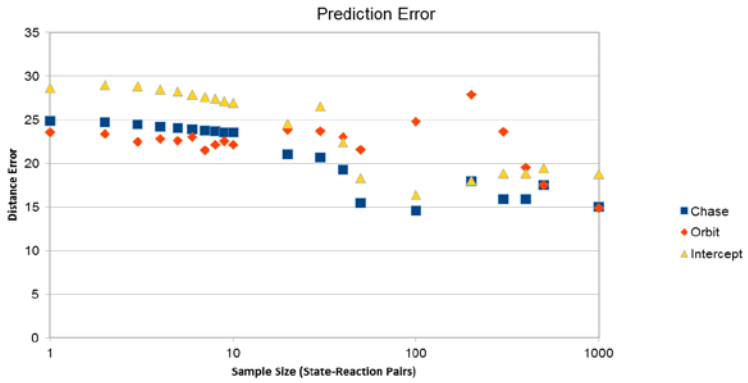

Figure 12: The distance between the predicted location of the enemy and its actual location

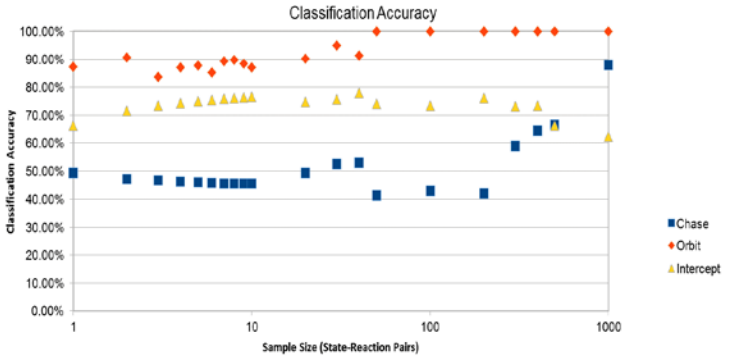

Figure 13: The percentage of correctly classified enemies

3.2.3 Coordination Module

Our approach is composed of three components: Quad-Tree decomposition, task allocation, and multi-layered coordination. The Quad-Tree is used to model the environment such that each quadrant in the tree has an *a priori* probability, provided through reconnaissance, that the corresponding area has one or more pieces of intel. By restricting the underlying representation of the problem to a Quad-Tree, large areas can be explored quickly using limited resources [19,20]. However, this approach is not restricted to Quad-Trees. The only requirement is that the underlying

representation of the environment must be decomposed into identifiable areas that can be used to model an *a priori* probability distribution of the intel locations. Therefore, our approach can be applied to alternative representations like topological, coverage, or occupancy maps.

Task allocation is performed using a utility-based algorithm centrally at the Home Base (HB). The centralized allocation occurs before the mission starts and with full knowledge of the drones and their capabilities, but with uncertain information about the environment. The decision concerning task allocations for the drones is modeled by the maximization of a utility function. The utility function considers the applicability of the sensor carried by the drone to acquire the Intel, the potential fuel consumption, and the time required to reach the intel. A max-flow algorithm is implemented on a K-Partite graph that maximizes the utility during task allocation.

For the task coordination, we design a three-layered coordination approach including: *Class Hierarchy, Message Passing, and Runner & CLI*. The first layer covers the Class Hierarchy that is used to separate the many tasks that a single drone has to service. The second layer covers the message passing protocol that drones use to communicate with each other and with the HB. The caveat to the message passing protocol is that – much like network protocols – there are many different layers of message passing – one for each class in Class Hierarchy. The Runner is the class responsible for executing each trial run and communicating with the Docker instance while the Command Line Interface (CLI) is used to configure the input arguments to the Runner such as which scenario will be the subject of the current run.

3.2.3.1 Layer1: Class Hierarchy

The Asset is responsible for managing the mutable fields of an ATE3 Asset such a fuel, motion, etc. Each Asset that is received from the ATE3 Docker instance is first converted to an immutable AssetProto, which is then passed to an instance of Asset. The AssetProto stores the immutable fields of the Asset such as the max fuel capacity, the max speed, etc. By doing so, the Asset instance can manage the mutable fields of the Asset while also ensuring that the immutable fields never violate the constraints of the Asset – e.g. an asset having more fuel than their max fuel capacity. In order to handle mutating the mutable fields, the Asset implements an update loop that takes an ATE3 AssetState. It then updates its own mutable state to deeply match that of the AssetState. This addresses our first requirement – syncing our local asset states with the Docker AssetStates.

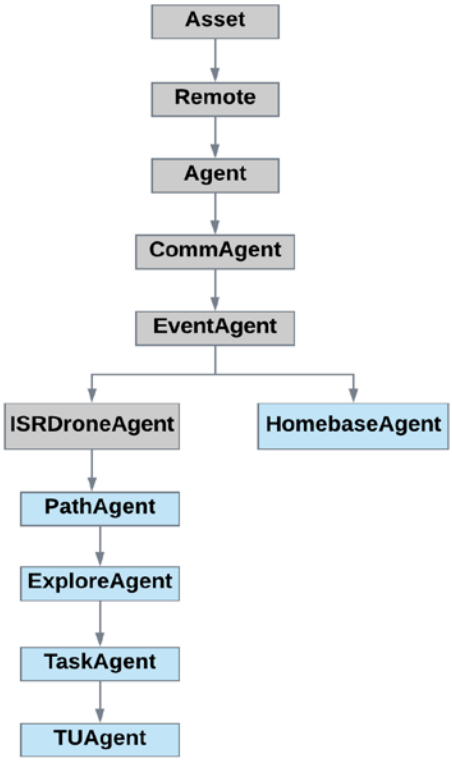

Figure 14: The class hierarchy for task coordination

Each non-abstract subclass in the class hierarchy overrides the update method while also calling the super instance of the method so that all levels of the hierarchy are able to update. This allows one level of the hierarchy to manage updating the local asset fields while another level of the hierarchy can manage checking for enemies within vision range.

The second class in the hierarchy is the Remote. A Remote is an Asset with an output loop that takes a builder instance of the ATE3 AgentIntent and updates the builder to include its own Intent. This loop is an abstract method. Each subclass will override this method while also calling the super instance of the method so that each class of the hierarchy will be able to configure the intent. This is done so that one class of the hierarchy can be responsible for the message intents, another class can be responsible for the event intents, while another class can be responsible for the move intent.

The next class in the hierarchy is the Agent class that serves as the root class for all of the implemented Agent types. The Agent class adds a Time requirement to the hierarchy as well. Agents exist to operate in the environment and execute a mission, so they must have a mission deadline. This is done by passing the Agent a TimeProto, which is then converted to a mutable Time instance for tracking how much Time the Agent has remaining in its mission.

Extending from the Agent class is the CommAgent class. This class is responsible for handling communications. In each update cycle, this class fetches messages from the inbox, parses those messages, and triggers a message incoming event, which can be hooked into by classes that are

lower on the hierarchy. This class also introduces a set of methods to manage a message queue that can also be employed by classes that are lower in the hierarchy. Then, during every loop, the CommAgent flushes all of the messages that have been queued to the Intent builder so that the messages can be passed on to their intended recipients.

The EventAgent is responsible for handling all of the events that are generated during execution such as New Intel, New Jammer, New AA, etc. These events need to be logged and sent to the Docker so that they can be recorded for the performance metrics tool that has been provided. In addition to logging events, the EventAgent also handles communicating events to other drones and to HB, and it handles logging events that are received as messages from other EventAgents.

There are two types of distinct EventAgents. The first is the ISRDroneAgent, which is responsible for the ISRDrone remote, and the second is the HomebaseAgent, which is responsible for the HB remote. The ISRDrones need additional functionality that the HB does not require such as exploring the environment and collecting intel. HB is responsible for updating the Docker for the performance metrics purposes.

The ISRDrone has its own hierarchy starting with the PathAgent. This agent is responsible for constructing and maintaining the physical path that the agent is to fly. Each path is represented as both an array of 2D Euclidean points that outline the physical path and as an array of 2D vectors that detail the trajectory of the path in terms of velocity. This is done so that the agent has a representation of the path that is adequate to send the Docker as a series of move intents. This agent is built using Predictive-RA*, which was explained in the previous section. Lastly, the PathAgent also exposes a series of hooks that relate to when conditions of the path-planner are violated. These conditions include when a new adversary enters visual range, when the PathAgent enters the visual range of an adversary, and when the PathAgent enters the weapon range of an adversary. These conditions trigger what is commonly referred to as replanning, and each of these generates a different replanning event that can be hooked into by classes that are implemented at lower levels of the hierarchy.

Extending from PathAgent is the ExploreAgent. The ExploreAgent is responsible for exploring the environment to locate more intel targets and learning more about the environment. Since every ally drone has the same visual range, the environment is represented using a Quad-tree. The Quad-tree is constructed from the root down so the root quad has the same dimensions as the environment. The construction ends when the dimensions of the leaf quads are no greater than two times the visual range of the ally drones. This construction was chosen because it implies that the location of a quad (its center point) is sufficient for observing the entire quad. To explore the entire environment, the ally drones need only visit every quad location. This approach also effectively maps the environment to a 2D Euclidean grid, which is easier to reason about. The leaf quads are then mapped to a set of explore tasks – one for each leaf quad. To explore the environment, the ExploreAgents must collaborate over the available tasks. This is done by first implementing *a ranking system* that the drones share. This ranking system determines which drone outranks other drones. Thus, if two drones attempt to take the same task, then the drone that is lower in the ranking order will abort the task and take another.

In limited communication cases, it is likely that more drones will attempt to explore the same quad unless the drones with lower ranks are informed en route to abort. To acquire an explore task,

an ExploreAgent chooses the closest explore task that is also available. An explore task is available if it is both unclaimed and incomplete. After claiming an explore task, the ExploreAgent immediately informs all other ExploreAgents and HB that it has claimed the task. The ExploreAgent is also responsible for sending other updates on its task, which will be explained more in the second layer of the approach.

The TaskAgent is very similar to the ExploreAgent except that it handles intel collection tasks and support tasks rather than exploration tasks. If any intel is known in advance, those intel are mapped to collection tasks. The TaskAgents then use the same ranking order that they would use to mitigate conflicts when exploring to mitigate conflicts when claiming collection tasks. Similar to the ExploreAgent, each TaskAgent claims the closest collection task that is also available. Available is defined as being unclaimed, incomplete, and having the same sensor type requirement as the drone that is attempting to collect it. Each TaskAgent handles reporting all updates about its collection task, which will be explained further in layer 2. In addition to collection tasks, TaskAgent can also attempt support tasks. Support tasks are generated when a drone is having difficulty collecting an intel. Difficulty is defined as being intercepted by an adversary while attempting to collect an intel. When a TaskAgent attempts to support, it will attempt to collect the intel task of the drone that it is supporting. In this way, each drone and its supporters are all collaborating to collect the same intel. If a TaskAgent has no collection tasks and no support tasks, then it behaves the same as an ExploreAgent.

The last agent implementation in the hierarchy is the TUAgent. The TUAgent is responsible for handling the state logic that directs the drone how to behave. The TUAgent has 11 states that it manages as a finite-state-machine FSM (Figure 15). These states are: DEAD, INACTIVE, STANDBY, EXPLORE, ACTIVE, INTERCEPT, HOMEBOUND, INVESTIGATE, PERSISTENT, DURESS, and JAMMED. The DEAD state indicates the drone has been killed either by being shot down by an adversary or by crashing after running out of fuel. The second state is INACTIVE which indicates the drone is alive, but is running out of time, fuel, or both. While operating in the environment, each drone will continuously check if it has enough time and fuel to return home. If it discerns that it only has enough time or fuel to return home, then it will do so and enter into the INACTIVE state. A drone that is INACTIVE aborts all of its tasks, returns home, and takes no other tasks for the remainder of the mission.

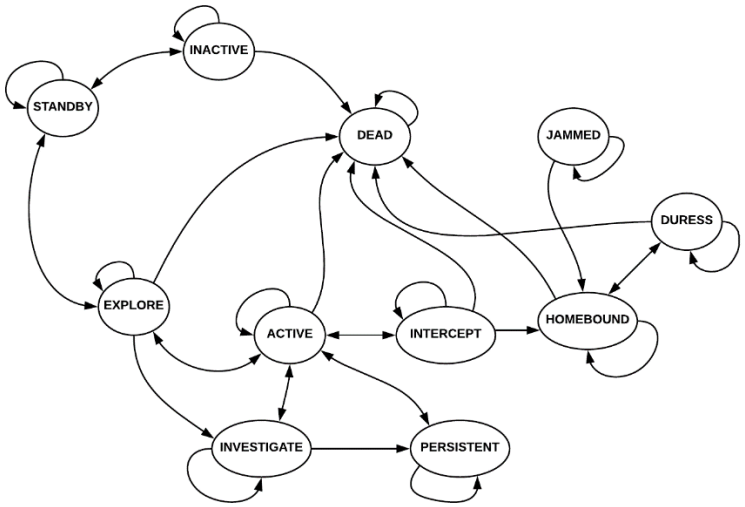

Figure 15: Finite-state machine for states of TUAgent

In STANDBY, a drone waits at HB until a new task becomes available. This state is rarely entered into because it requires there to be no tasks left available. This only occurs when the entire environment has been explored and when all of the intel have been collected. As such, when all of the drones are either DEAD, INACTIVE, or on STANDBY, the mission can be terminated. The next state is the EXPLORE state. A drone is in the EXPLORE state if there are no collection tasks or support tasks available, but there are explore tasks that are available. While in the EXPLORE state, a drone is actively claiming explore tasks, completing explore tasks, and reporting completed explore tasks. However, if there are any collection tasks or support tasks, then the drone will exit the EXPLORE state and enter the ACTIVE state. While in the ACTIVE state, a drone is actively executing a path planning towards its intended intel objective and replanning to avoid adversaries along the way. If the drone can successfully reach its intel objective and begins collecting intel, then it enters the INTERCEPT state. Once in the INTERCEPT state, a drone hovers until it has collected the target intel. After collecting the target intel, the drone can transition to the ACTIVE state if there are more intel to collect, the EXPLORE state if there are no intel to collect, but there are explore tasks to explore, or the drone can enter the HOMEBOUND state. A drone enters the HOMEBOUND state when it is returning home and has no explore tasks, collection tasks, or support tasks to claim. Once the drone reaches HB, it transitions to the STANDBY state.

The other three states are special states that occur when a drone is either seen or threatened by an adversary. The INVESTIGATE state occurs when a drone has been seen, but is not being chased. In this state, the drone will attempt to reach its intel objective while also avoiding the weapon range of nearby adversaries. As a secondary objective, it will also attempt to escape the visual range of all nearby adversaries, but not at the cost of losing intel. This is done because not all adversaries are a direct threat. For example, adversaries that don't chase, those without weapons, and those that cannot move are non-threatening. As an additional component of the INVESTIGATE state, the drone is attempting to determine if it needs support. A drone that transitions to the INVESTIGATE state doesn't immediately call for support. Instead, it first attempts to collect the intel, and if it is chased or intercepted in the process, it then calls for support. The reason for this delay is due to the last component of the INVESTIGATE state. The drone is

investigating how many adversaries are guarding its intel objective. This informs the drone how many other drones it should request to support. In general, the number of adversaries that are spotted during investigation are the number of additional drones that can support the intel collection task. However, a drone must have the same sensor type as the intel objective. This type is not always known, but it can be communicated about. A drone that is investigating will not include static adversaries when reporting how much support it needs. Also, if the drone determines that there is no way to collect an intel objective because of a static adversary, it will cease attempting to collect that intel and will broadcast a global cancel so that no other drone attempts to collect that intel either.

While attempting to INVESTIGATE, a drone can be chased or intercepted. This occurs when an adversary is actively trying to intercept our drone or when the drone has already reached weapon range of an adversary. In either of these cases, the drone transitions to the PERSISTENT state. In the PERSISTENT state, the drone switches its priorities so that its primary priority is to avoid nearby adversaries and its secondary priority is to collect its intel. This is done so that the drone will be allowed to flee the intel objective for its safety before trying again. If a drone transitions into the PERSISTENT state, it will ask for support. The DURESS state mirrors that of the PERSISTENT state. The DURESS state occurs when a drone is actively trying to reach HB, but is being chased or has been intercepted. In these cases, the drone makes its first priority escaping the adversary and its second objective reaching HB. However, a drone that is in the INACTIVE state cannot transition to the DURESS state as it is assumed that the drone doesn't even have enough time or fuel for evasive maneuvers so the drone must risk being shot down or it will most surely crash.

We refer to the three states: INVESTIGATE, PERSISTENT, and DURESS as the three special states because they are the only three states where the drone can cease using its Predictive-RA*. When being chased or intercepted, it is not often the best approach to calculate a trajectory of escape as any trajectory that is generated will likely be replanned after the next step. In these cases, rather than using Predictive-RA*, the drone employs a *Vector Field* to dictate motion. The vector field is a representation of the environment and its assets as push- and pull-forces. Each adversary represents a push-force. The push-force is translated according to the current velocity of the adversary and its relative distance from the drone. The objective of the drone represents a pull-force. The pull-force is also translated by the distance the objective is away from the drone. One other advantage of representing the environment in this fashion is that it allows the drone to treat its objective as a sink. In such cases when the drone is actively avoiding adversaries, but is also collecting intel, it will attempt to stay within collection range while avoiding the adversary. Hence, the drone will typically only cease collection if the intel target would come within weapon range. The only difference being that in the DURESS state, HB is treated as the target.

The final state is the JAMMED state. The JAMMED state occurs when a TUAgent has a message that it needs to share with other TUAgents or with the HomebaseAgent, but is currently within jamming range. In this state, it is the top priority to leave jamming range. However, a TUAgent won't always know where the jammer is so it won't know the best direction to take to leave jamming range quickly. As such, a TUAgent in the JAMMED state will head back towards HB. In this way, it can share its messages with the HomebaseAgent to guarantee that the messages

were received by at least one other agent. The HB also carries ECCM so a TUAgent will be guaranteed to send if it can get close enough to HB. A TUAgent can transition out of the JAMMED state once it has shared its messages with the HomebaseAgent. Because of this, the HomebaseAgent can also be a useful agent to share the most current information. This will be discussed further in the second layer.

3.2.3.2 Layer2: Message Passing

To allow drones to collaborate in a decentralized fashion, a number of message passing protocols were defined. In total, there are four message passing protocols that have been created. The first is for message passing between EventAgents, which utilizes the DiscoveryEventMessage and the IntelEventMessage. The second is for message passing between ExploreAgents and with the HomebaseAgent, which utilizes the QuadControlMessage. The third is for message passing between TaskAgents and with the HomebaseAgent, which utilizes the TaskControlMessage. Finally, the fourth is for message passing between TUAgents and with the HomebaseAgent, which utilizes the AgentControlMessage.

1) EventAgent Communication Protocol:

The principal responsibility of every EventAgent is to register events and communicate those events to the other EventAgents so that they register those events as well. The events that must be registered include the New Intel event, the New Jammer event, the New AA event, and the Lost Drone event. When a new intel is discovered, a new jammer is discovered, a new AA is discovered, or a lost drone is discovered, then the EventAgent sends a DiscoveryEventMessage to the other EventAgents to report it. It is often the case that when a new intel is discovered, the sensor type

requirement for that intel is not known. If it becomes known, then an IntelEventMessage is broadcasted to report the sensor type. In addition to reporting the known intel type, the IntelEventMessage can also be used to broadcast an inverted sensor type. This inverted sensor type indicates that this intel is no longer available to any drone that has the same sensor type as the inverted sensor type. Any drone with a different sensor type may still attempt to claim the intel. It can also occur that a drone is not able to come close enough to the intel to determine the sensor type or if it needs to report an inverted sensor type. As such, the first sensor type that is reported for an intel – if the sensor type is not already known – is the unknown sensor type. An intel with an unknown sensor type is available to all drones. The final purpose of the IntelEventMessage is to report the intel string of collected intel. Any drone that has an intel string to report that is also within jamming range will attempt to escape jamming range in order to report the collected intel string.

Since drones can also be jammed, it is also important to consider what would happen if the message is not received. Assuming that an EventAgent is jammed, it would not be able to receive or broadcast DiscoveryEventMessages or IntelEventMessages. These messages do not impact the decision making of the drone nor its tasks. As such, these messages are secondary to whatever else the drone is currently doing with the only two exceptions being for new intel and intel strings. If a new intel is spotted, then that intel objective must be reported so that it can be collected. If a drone has an intel string that it has collected, it must find a way to report that intel string.

2) ExploreAgents Communication Protocol:

To collaborate on exploration tasks, ExploreAgents must be able to communicate about exploration tasks. The QuadControlMessage is used to communicate which exploration tasks have been claimed, aborted, or completed. If an exploration task is claimed by multiple ExploreAgents, then the ExploreAgents with the lowest ranks abort the exploration task. This can be tacitly done as the other drones which received both claim messages know to use the claim message from the ExploreAgent with the greater rank. Ranks are shared while the drones are still at HB and do not change. If a drone has claimed an exploration task, but cannot complete the task or has chosen a collection task or support task over the exploration task, then it must abort its current exploration task. It will send a QuadControlMessage with the ABORT signal to inform the other ExploreAgents that they can now claim the aborted exploration task. Lastly, if an ExploreAgent is successful in exploring its objective quad, then it will broadcast a COMPLETE signal using the QuadControlMessage. This informs the other ExploreAgents that the claimed task has now been completed.

In the event that an ExploreAgent is jammed, then it will not be able to send or receive QuadControlMessages. If an ExploreAgent does not receive a CLAIM signal, then it might attempt to explore the same task as another ExploreAgent. This leads to redundancy that results in wasted time and fuel, but ultimately not a huge expense. Redundantly exploring the same area can also provide drones who couldn't communicate the same knowledge about the same environment. This can even potentially aid in the collection of intel. Overall, it can often cost more resources to leave jamming range to send a CLAIM signal than it would cost for another drone to explore that quad redundantly since ExploreAgents explore the quads nearest them and jamming ranges can be rather

large. The other two messages are the ABORT signal and the COMPLETE signal. The ABORT signal can have an impact on the results of the exploration because if other drones do not receive the ABORT signal, then they will never explore that piece of the environment as they will never know that the task has become available for claim. However, ExploreAgents only ABORT in the event that they have claimed a collection or a support task. As such, it is of greater priority for them to proceed with that task than it is for them to discover a means of communicating the ABORT. As it turns out, they can typically communicate the ABORT at a later time after they have completed their other tasks. The COMPLETE is of no impact if it is not received as any task that is completed was previously claimed. A claimed task cannot be claimed by another ExploreAgent so a completed task will not be redundantly visited.

3) TaskAgents Communication Protocol:

TaskAgents use TaskControlMessages to communicate about the state of current tasks. There are eight signals that can be sent using a TaskControlMessage: OPEN, CLAIM, CLOSE, COMPLETE, ABORT, CANCEL, UPDATE, and SUPPORT. A message is sent with the OPEN signal when a new intel objective has been detected and converted to a task. In some sense, this is redundant because any drone that received the intel DiscoveryEventMessage would also know to convert the new intel objective to a task. However, we leave the management of tasks to the TaskAgent so the TaskAgent signals when a Task should be opened. A task can be opened only when the task is currently available to the TaskAgent. This means that the sensor type requirement of the task is either unknown, not the inverted sensor type of the TaskAgent, or is the same sensor type as the TaskAgent.

Once a task has been opened, it can be claimed. If a TaskAgent claims a task, then they send out the CLAIM signal to inform the other TaskAgents and the HomebaseAgent that the task has been claimed. If more than one drone sends out a CLAIM signal for the same intel objective, then the drone with the greatest rank will keep the task and the others will abort. This can be done silently as the other receiving drones know which drone will keep the claim. If a task has been claimed, then no other TaskAgent will attempt to collect that intel objective unless they are supporting. If a task has been claimed, then the TaskAgent who claimed the task can also signal a COMPLETE, an ABORT, or a CANCEL. If the drone has successfully collected the intel string from its objective, then it will signal a COMPLETE that informs the other drones that it has completed its task. Otherwise, if the TaskAgent could not complete the task and had to give up the task, then it will signal an ABORT, which shows that the task should be claimed by another TaskAgent. In certain edge cases, there is no way to collect an intel objective. In such cases, the drone who claimed the intel task can signal a CANCEL, which informs the other TaskAgents that this task cannot be completed.

The last two signals are used to communicate information about a task. The UPDATE signal is sent whenever the sensor type requirement of an intel becomes known or if adversaries are spotted near an intel objective during investigation. The SUPPORT signal is used to both ask for support and notify that support is on its way. In the event that a TaskAgent believes that it needs aid for its current assignment, it will signal a SUPPORT with how much SUPPORT it needs. TaskAgents that receive the SUPPORT signal and can support then respond with a SUPPORT signal to notify

the original recipient that they are supporting. Same as with claiming tasks, if too many TaskAgents signal a support, then those with the lowest rank will abandon.

In case of limited or jammed communications, TaskAgent is embedded with a simple strategy to determine what will happen in the event that signals are not sent or cannot be received. In the case that a TaskAgent has discovered intel, but cannot send an OPEN signal, then that intel will remain unknown to the remaining TaskAgents. If this intel can be collected by the TaskAgent who discovered it, then they can proceed with attempting to collect the intel without moving out of jamming range to send the OPEN signal. If they do so, it is possible that they would be shot down or crash without collecting the intel or signaling other TaskAgents about the intel, but if they could not signal the OPEN, then they also couldn't signal that the quad had been explored. As such, other TaskAgents will explore this area and will rediscover the intel objective. However, if they cannot collect the intel objective, then they should make every effort to broadcast the OPEN signal so that other TaskAgents might be able to make the attempt to collect the intel objective.

In addition to being prohibited from sending an OPEN signal when they get jammed, TaskAgents can also be prohibited from sending the CLAIM, COMPLETE, ABORT, and CANCEL signals. If a drone is unable to send the CLAIM signal, then it is possible that another drone or drones might attempt to collect the same intel objective. This would be redundant, but it would also increase the chances of collecting the intel objective. As such, it is not considered a priority to leave jamming range to send this signal. COMPLETE and CANCEL are two other signals that are not prioritized. If a COMPLETE signal is not sent, then no other drones will attempt to collect the intel objective as it has already been claimed and is therefore not available. However, it is of top priority that collected intel strings be broadcasted to other TaskAgents and to HB so drones will leave jamming range to communicate the intel string which is equivalent to leaving jamming range to communicate a COMPLETE signal. Similarly, a task that has been canceled won't be attempted because a task must first be claimed before it can be canceled. Another important signal to communicate about is the ABORT signal. If an ABORT signal is not sent, then other TaskAgents will not try to claim it. As such, any drone who is jammed must exit jamming range to broadcast an ABORT signal so that another drone may claim the task.

The last two signals are the UPDATE and SUPPORT signals. The UPDATE signal is used to UPDATE the sensor type requirement of an intel objective and is also used to update the number of adversaries near an objective. In the event that an UPDATE signal is not received, and an ABORT signal is sent that is received, then it is possible that a drone who does not have the correct sensor type requirement for the intel objective would attempt the objective. However, if a TaskAgent can receive the ABORT signal, then they can also be updated with the correct sensor type as well. Therefore, it is not a priority to leave jamming range to send an UPDATE signal for this reason. It is also not a priority to leave jamming range to send an UPDATE signal to update the number of adversaries near an intel objective. Adversaries are mobile and the number of adversaries near an intel objective is constantly changing. As such, it is likely that by the time a TaskAgent left jamming range to send the UPDATE signal, the number of adversaries around the objective would have already changed. While it may not be a priority to send an UPDATE signal, it is a priority to send a SUPPORT signal. If a TaskAgent sends a SUPPORT signal, it is because they need other TaskAgents to aid them in order to collect the intel. If this signal is not sent, then

no other TaskAgents will aid. As such, the TaskAgent must leave jamming range to broadcast the SUPPORT signal so that other TaskAgents will support. The TaskAgent can also wait for SUPPORT responses to confirm that other drones are supporting.

4) TUAgents Communication Protocol:

TUAgents communicate by passing AgentControlMessages. The purpose of this communication layer is to synchronize data between the sharing agents. and with the HomebaseAgent. When two agents can share an AgentControlMessage, they can then directly message one another the state of their explore tasks, collection tasks, and support tasks. These states are recorded as a series of updates that were made to their tasks, which includes opening tasks, claiming tasks, completing tasks, aborting tasks, etc. There can be conflicts when two or more task agents both claimed the same task, but the two TUAgents can use the ranking order to resolve the conflict. The importance of this protocol is that it allows a TUAgent, which is aware of available tasks or unavailable tasks, to share those tasks with another TUAgent. Thus, the recipient will have the opportunity to attempt tasks that they were unaware are available and will not attempt tasks that are unavailable. This message is not sent as a response to an event, but rather is something that occurs whenever a TUAgent can share messages with another TUAgent or with the HomebaseAgent. As such, a drone that is jammed doesn't need to exit jamming range in order to send an AgentControlMessage. Eventually, the drone will run out of tasks and will head to homebase and then have the opportunity to share an AgentControlMessage.

3.2.3.3 Layer3: Runner & CLI

The Runner is responsible for generating the pre-execution information such as the rankings of the agents, updating them, and communicating with the Docker instance. Each run of a scenario can be configured by providing input parameters that are handled by the CLI. By default, the Runner is set to ignore all intel targets and adversaries that are available in the config file. By doing so, no agent will have any information on their environment requiring that the agents work together to explore their environment to discover the intel objectives. This default mode can be overwritten by providing the --intel_known command. If this command is provided, then all intel objectives will be made known to the agents prior to mission start. The agents won't assume that this set of objectives is complete so they will still explore their environment after all known objectives have been discovered. They will be provided the locations of the objectives as written in the config. As such, to provide only approximate locations, a recon file can be provided to the Runner instead of a config file. This will not interfere with the Runner in any way. If it is desired that only some, but not all, of the intel objectives are made known to the agents, then in addition to the --intel_known command, the --pop_ups command can also be used to pass the GUIDs of the intel objectives that should not be made available to the agents. The input to the --pop_ups command is a comma-separate, but not space separated, list of GUIDs.

The --red_vision, --red_weapon, --red_speed, and --blue_speed commands can be used to reconfigure agent behavior and expectations on the fly. These should be set to reflect the agent's expectations of their adversaries. They do not have to be set to the actual values of the adversaries' vision range, weapon range, and speed. The Runner will not parse the config file to inspect these values. The last commands of interest are the --alpha and --points commands. These commands are used to configure the input parameters to the path planner. Our path planner is based on RA* so the alpha parameter is used as a heuristic to weight paths while the points parameter is meant to reflect how many points are required to collect an intel. This is used by agents to determine how long they should attempt to collect an intel objective. The points command does not have to be exact. However, if the points command is less than the actual points of an intel objective, then it is possible that a drone will abandon if it has other tasks that it can claim with more points.

4.0 Results and Discussion

To evaluate the resiliency of our solution, we ran it against a total of three diverse scenarios given by the AFRL/RI MIMFA team. Each scenario (S1 through S3) has a permissive version and an A2AD (non-permissive, contested) version. In the permissive version of a scenario, it is relatively easy to reach maximum possible levels of performance. The A2AD version of the scenario (essentially a shift or extension from the corresponding permissive version) introduces added difficulty in terms of uncertainty (number and locations of intel or anti-air), a contested communications model due to jamming, and increased potential for asset loss. S1P refers to Scenario 1 Permissive. S1A refers to Scenario 1 A2AD and so on.

4.1 A2AD Event List

This is the list of potential A2AD events that could occur in any of the A2AD versions of the scenarios:

- Limited comm range. This is implemented with a gradient level of jamming/comm power from the center of the jammer/transmitter
- Inability to refuel. The drone starts with a limited quantity of fuel and is unable to refuel during the mission time.
- Increased jamming range (and/or effectiveness). Effective jamming range may cover 50% to 80% of the map and Jammers would remain stationary.
- Increased/decreased speed of Mobile AAs on an asset-by-asset basis.
- Inexact intel target positioning. Ability to prevent the drones from knowing adversarial asset exact or general locations beforehand. This includes Mobile AAs, Jammers, and intel assets.
- Reduced or increased visual range of adversarial assets on an asset-by-asset basis.

4.2 Objective Function

The following are the metrics of interest and their relative weight/importance

Table 2: The performance metrics

Variable	Relative Weight (%)	Description
IAPI	30	% identified positions of Red Intel Assets
ISVC	25	% Intel total value captured
SM	15	% of total minutes spare after mission completion
BFS	15	% of Blue Force Assets Survived
AAPI	10	% identified positions of Red Anti-Air
JAPI	5	% identified positions of Red Jammers

4.3 Scenarios

4.3.1 Scenario 1

Time Limit: 30 minutes of simulation time.
Permissive: The environment size is 6x6 km with the home base at the center. There are 15 Blue Force, all have the same sensor type. The Red Force are 5 Mobile AAs, 5 Jammers, and 20 intel sites randomly placed in the environment. All Red Force asset types, intel types, and locations are known beforehand. All 20 intel sites have a single intel item with a value of 1.0.
A2AD Event: Mobile AAs are increased to 25 assets. The Blue Force communication power reduced to half. None of the intel site information is known beforehand.
Results: By computing details about intel collection, information dissemination, and map coverage using the FarCo calculator; we were able to precisely find the performance scores for both permissive and A2AD versions. As shown in Table 4, the performance score for S1P is 84.12% while S1A's performance score is 81.07%. Hence, the resiliency of our solution in Scenario 1 is 96.4%.

Table 3: Performance metrics for Scenario 1

Variable	S1P	S1A
IAPI	100	100
ISVC	100	100
SM	0.805	0.472
BFS	93.33	73.33
AAPI	100	100
JAPI	100	100
Score	**84.12%**	**81.07%**

4.3.2 Scenario 2

Time Limit: 45 minutes of simulation time.
Permissive: The environment size is 10x10 km. Blue Force home base is situated at extreme upper-left. There are 30 Blue Force assets with sensor type A, 10 Blue Force assets with sensor type B. The Red Force are 20 Mobile AAs, 10 Jammers, 40 intel sites, spread randomly in NE, SE, and SW quadrants. All Red Force asset types, intel types, and locations are known

beforehand. Half of the intel sites are of type A and the other half are of type B. Each intel site contains three intel items. Type A intel items are valued at 1.0 and Type B intel items are valued at 2.0.

A2AD Event: None of the intel site information is known beforehand. Red Force increased to 60 Mobile AA assets. Half of the Red Force AA sites are able to move 3% faster than the default Blue Force speed.

Results: The performance score for S2P is 71.4% while S2A's performance score is 59.1%. Hence, the resiliency of our solution in Scenario 2 is 83% (Table 4).

=

Table 4: Performance metrics for Scenario 2

Variable	S2P	S2A
IAPI	97.5	85
ISVC	67.77	49.44
SM	0.537	2.759
BFS	77.5	60
AAPI	90	83.33
JAPI	90	70
Score	**71.4%**	**59.1%**

4.3.3 Scenario 3

Time Limit: 75 minutes of simulation time.

Permissive: The environment size is 16x16 km. Blue Force home base is situated at extreme upper-right. There are 30 Blue Force assets with sensor type A, 30 Blue Force assets with sensor type D, and 10 Blue Force assets with sensor type E. The Red Force are 40 Mobile AAs, 10 Jammers, 100 intel sites, spread randomly in NW, SE, and SW quadrants. All Red Force asset types, intel types, and locations are known beforehand. 33 intel sites are of type A, 33 are of type D, and 34 are of type C. Each intel site has four intel items, each of which are valued at 1.0.

A2AD Event: None of the Red Force information is known beforehand and Blue Force assets are unable to refuel. Half of the Blue Force is reduced to 50% of fuel capacity and Red Force increased to 100 Mobile AA assets. Jamming power is increased 20%.

Results: AS displayed in Table 5, the performance score for S3P is 49.94% while S3A's performance score is 24.735%. Hence, the resiliency of our solution in Scenario 3 is 49.4%.

Table 5: Performance metrics for Scenario 3

Variable	S3P	S3A
IAPI	72	8
ISVC	59.5	4.5
SM	0.655	82.233

BFS	32.5	52.5
AAPI	60	10
JAPI	50	0
Score	**49.94%**	**24.735%**

5.0 Conclusion

We have done a tremendous amount of work over the duration of this project. Beginning with an intuition about the relationship between the dynamic environment and the robustness of the mission plans, we developed a path planning algorithm called Robustness A* (RA*) to dynamically and flexibly determine the risk avoidance based on the latest information about the environment and mission constraints. It addresses the "reach-while-avoid-when-possible" path planning problem by using the MTL robustness theory. Building on RA*, we developed a learning framework using IRL, MDP, and KF to predict the behaviors carried by enemy agents as a reaction to our drone's action. The P-MTL is used to conduct temporal logic reasoning over probabilistic and predicted states of KF.

We did not stop there, however. We developed an intuitive approach for the task collaboration and coordination between the ally drones using *a priori* reconnaissance about the task and risk distributions. We used the Quad-Tree and K-Partite graph to assign tasks in centralized and decentralized fashion. The Quad-Tree is utilized to model the uncertainty in Intel and enemies locations during the mission time. Then, the tree is used by the K-Partite graph to compute the utility functions for drones and find their assignments maximizing the mission goal of collecting as many Intel as possible. The initial task assignments are centrally generated before the mission gets started then each drone can maintain its local copies of the Quad-Tree and graph to assign itself to other tasks. The drones communicate their information for coordination purposes. Combined, we showed a super additive effect when the coordination components and policy components work together.

6.0 Future Work

Although we have had substantial achievements during this project, there is still a great deal of work to be completed to understand the long-term implications of this research. First and foremost, we would like to extend our current mission planning solution into cooperative, distributed planning which is a complex problem where each system develops an individual, local plan that is refined while coordinating with other systems to avoid potential conflicts and use cooperative opportunities for improved completion of mission objectives. We think that it will make our results much more useful. We would also like to study developing formal representation and reasoning techniques to weigh the impact that changes to the environment have on the value of information during the formation and execution of a plan.

We would also like to further explore using the Generative Adversarial Networks (GANs) to directly generate the distributions of states and actions of the observed enemy without the need to learn the reward functions in an offline training stage. This technique may help in identifying complex behaviors of enemies to increase the robustness of the drone's plans.

References

[1] A. Marshall, S. Alqahtani, A. Ridgway, C. Walter, R. Gamble, and R. Mailler, "Combining coordination with usage policies to improve mission scheduling resilience," in *2015 Resilience Week (RWS)*, 2015, pp. 1-6.

[2] M. Yokoo, T. Ishida, and K. Kuwabara, *Distributed constraint satisfaction for DAI problems*. 1990.

[3] M. Yokoo and E. Durfee, *Distributed Constraint Optimization as a Formal Model of Partially Adversarial Cooperation*. 1996.

[4] S. E. Conry, K. Kuwabara, V. R. Lesser, and R. A. Meyer, "Multistage negotiation for distributed constraint satisfaction," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 6, pp. 1462-1477, 1991.

[5] K. Sycara, S. F. Roth, N. Sadeh, and M. S. Fox, "Distributed constrained heuristic search," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 6, pp. 1446-1461, 1991.

[6] V. R. Lesser and D. D. Corkill, "The distributed vehicle monitoring testbed: a tool for investigating distributed problem solving networks," in *Readings from the AI magazine*, E. Robert, Ed.: American Association for Artificial Intelligence, 1988, pp. 69-85.

[7] R. Mailler, "Comparing two approaches to dynamic, distributed constraint satisfaction," presented at the Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, The Netherlands, 2005.

[8] R. Verfaillie, and N. Jussien, "Constraint Solving in Uncertain and Dynamic Environments: A Survey," *Constraints*, vol. 10, no. 3, pp. 253-281, 2005.

[9] G. E. Fainekos and G. J. Pappas, "Robustness of temporal logic specifications," presented at the Proceedings of the First combined international conference on Formal Approaches to Software Testing and Runtime Verification, Seattle, WA, 2006.

[10] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255-299, 1990.

[11] A. Dokhanchi, B. Hoxha, and G. Fainekos, "On-Line Monitoring for Temporal Logic Robustness," Cham, 2014, pp. 231-246: Springer International Publishing.

[12] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley Inc., 1994, p. 672.

[13] S. Alqahtani, I. Riley, S. Taylor, R. Gamble, and R. Mailler, "MTL Robustness for Path Planning with A*," presented at the Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, Stockholm, Sweden, 2018.

[14] A. Y. Ng and S. J. Russell, "Algorithms for Inverse Reinforcement Learning," presented at the Proceedings of the Seventeenth International Conference on Machine Learning, 2000.

[15] B. Piot, M. Geist, and O. Pietquin, "Bridging the Gap Between Imitation Learning and Inverse Reinforcement Learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 8, pp. 1814-1826, 2017.

[16] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35-45, 1960.

[17] S. Alqahtani, S. Taylor, I. Riley, R. Gamble, and R. Mailler, "Predictive Path Planning Algorithm Using Kalman Filters and MTL Robustness," in *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2018, pp. 1-7.

[18] M. SANKARAN, "Approximations to the non-central chi-square distribution," *Biometrika*, vol. 50, no. 1-2, pp. 199-204, 1963.

[19] Y. Zhang, K. Kim, and G. Fainekos, "DisCoF: Cooperative Pathfinding in Distributed Systems with Limited Sensing and Communication Range," Tokyo, 2016, pp. 325-340: Springer Japan.

[20] M. Peasgood, C. M. Clark, and J. McPhee, "A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps," *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 283-292, 2008.

List of Acronyms

AFRL – Air Force Research Labs

ATE^2 - Autonomy Test & Evaluation Environment

CLI - Command Line Interface

CM - Coordination Module

DCOP - Distributed Constraint Optimization Problem

DCSP - Distributed Constraint Satisfaction Problem

DynDCOP - Dynamic, Distributed Constraint Optimization Problem ECCM – Electronic Counter-Counter Measure

GAN – Generative Adversarial Network

GTC - geo-temporal constraints

HB - Home Base

IRL - Inverse Reinforcement Learning

ISR - Intelligence, Surveillance, and Reconnaissance

JR - Jamming Range

KF - Kalman Filter

MDP - Markov Decision Process

MEM - Mission Execution Module

MPM - Mission Planner Module

MTL - Metric Temporal Logic

PLF - Pattern Learning Framework

PLM - Pattern Learning Module

RA* - Robust A*

RHS - Robust Heuristic Search

UAV - Unmanned Aerial Vehicle

VR - Visual Range

WR - Weapon Range